

(10) Patent No.: US 6,389,433 B1
(45) Date of Patent: *May 14, 2002

- | | | |
|----|-------------|--------|
| WO | WO 99/12098 | 3/1999 |
| WO | WO 99/21082 | 4/1999 |

OTHER PUBLICATIONS

- LaLonde, Ken, "Batch daemon—README", UNIX Batch Comm. Association, University of Toronto, pp. 1–3 (Feb. 27, 1997), <http://ftp.cs.toronto.edu/pub/batch.tar.z> printed Dec. 8, 2000.
- Steere et al., "A Feedback-driven Proportion Allocator for Real-Rate Scheduling", *Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, USENIX Association, pp. 145–158 (1999).

- * cited by examiner

- Primary Examiner*—Hosain T. Alam

This patent is subject to a terminal disclaimer.

Assistant Examiner—Cam-Y Truong

(74) Attorney, Agent, or Firm—Michalik & Wylie, PLLC

(57) **ABSTRACT**

- (21) Appl. No.: 09/354,660

- (22) Filed: Jul. 16, 1999

- (51) Int. Cl.
- ⁷
- G06F 12/00

- (52) U.S. Cl. 707/205; 707/200; 707/201;
707/202; 707/203; 707/204; 707/3; 707/1;
707/10

- (58) **Field of Search** 707/1, 3, 10, 200-205

(56) **References Cited**

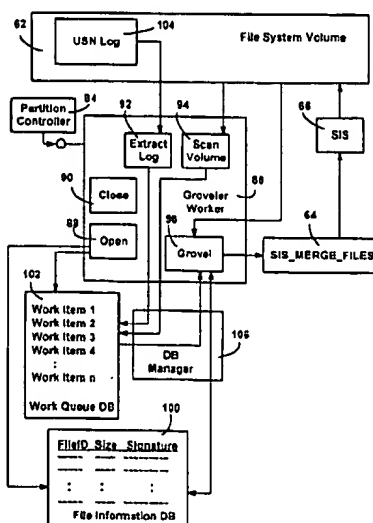
U.S. PATENT DOCUMENTS

5,410,667	A	4/1995	Belsan et al.	
5,706,510	A	1/1998	Burgoon	
5,778,384	A *	7/1998	Provino et al.	707/200
5,778,395	A *	7/1998	Whiting et al.	707/204
5,907,673	A *	5/1999	Hirayama et al.	395/182.14
5,918,229	A	6/1999	Davis et al.	
6,185,574	B1 *	2/2001	Howard et al.	707/200

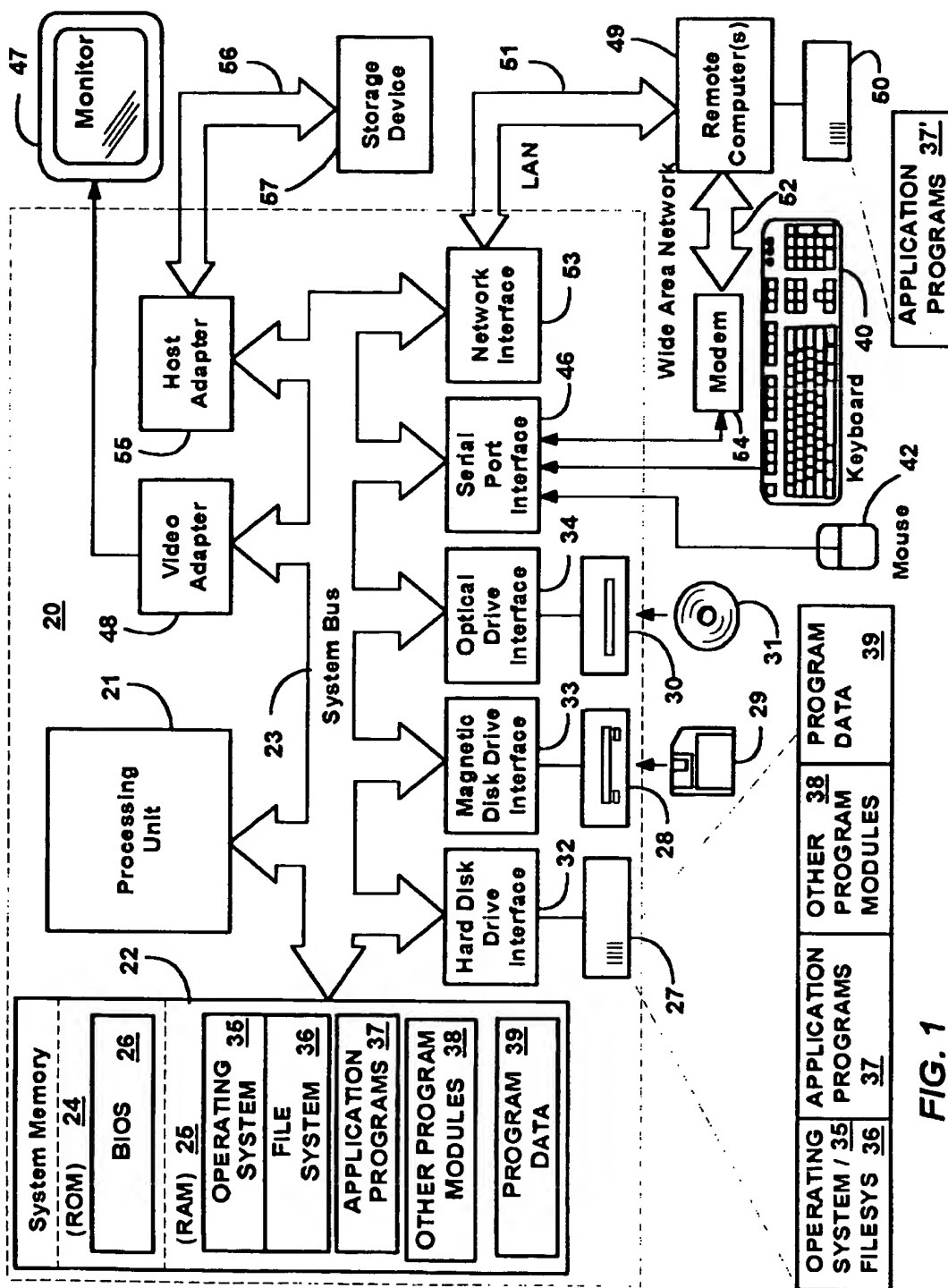
FOREIGN PATENT DOCUMENTS

EP	0 774 715 A1	5/1997
WO	WO 99/09480	2/1999

41 Claims, 22 Drawing Sheets



finding duplicate
files



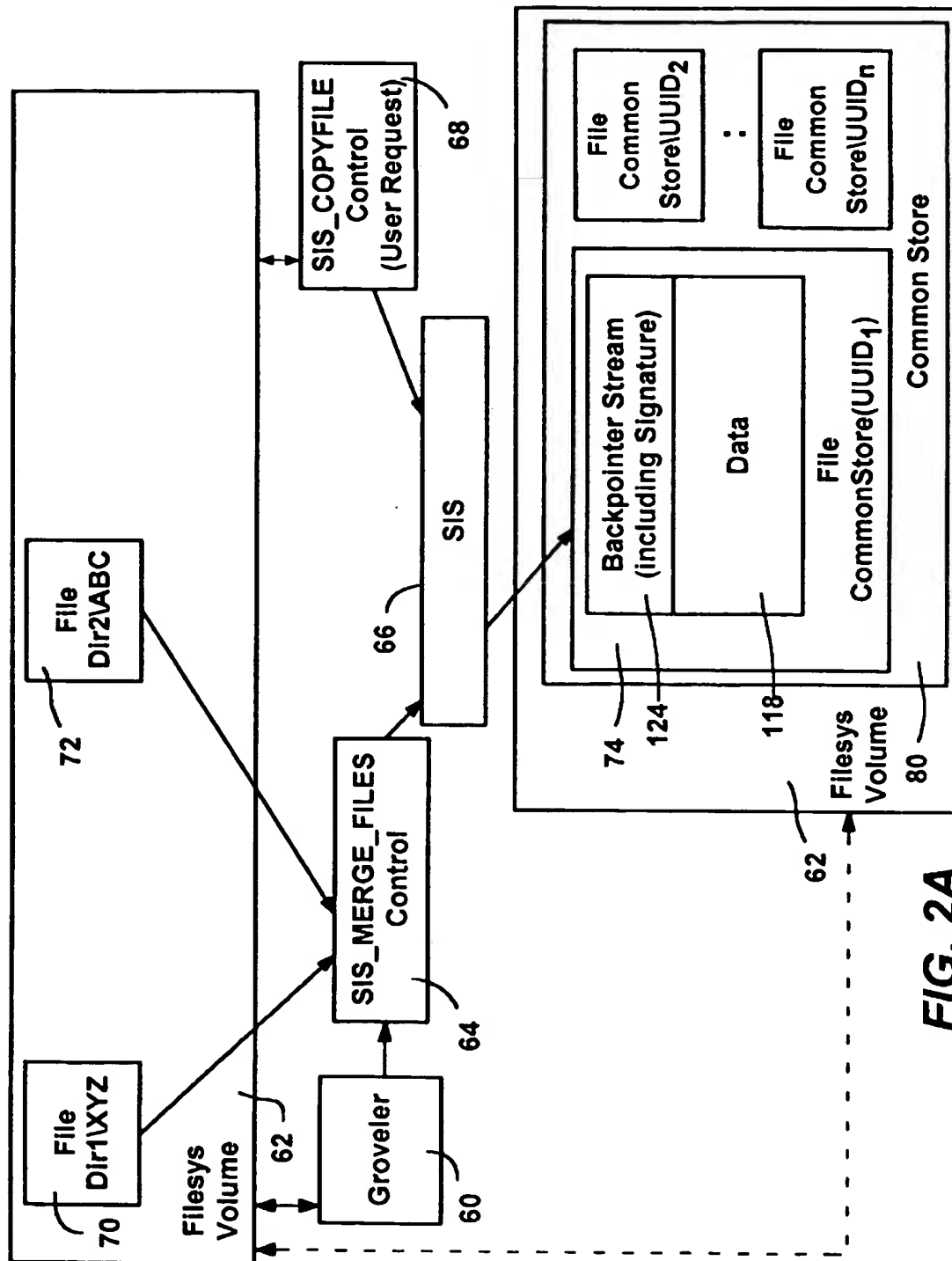
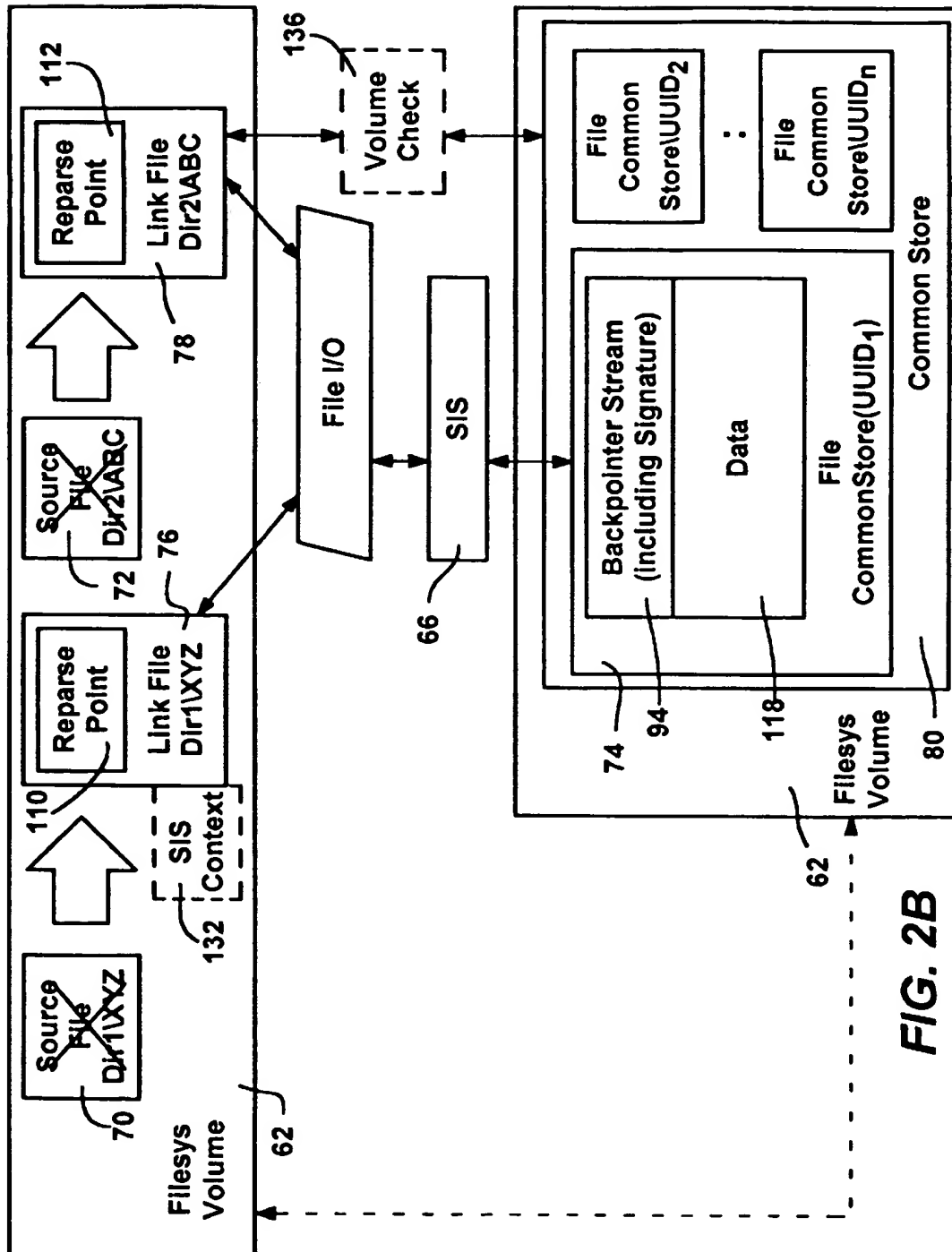
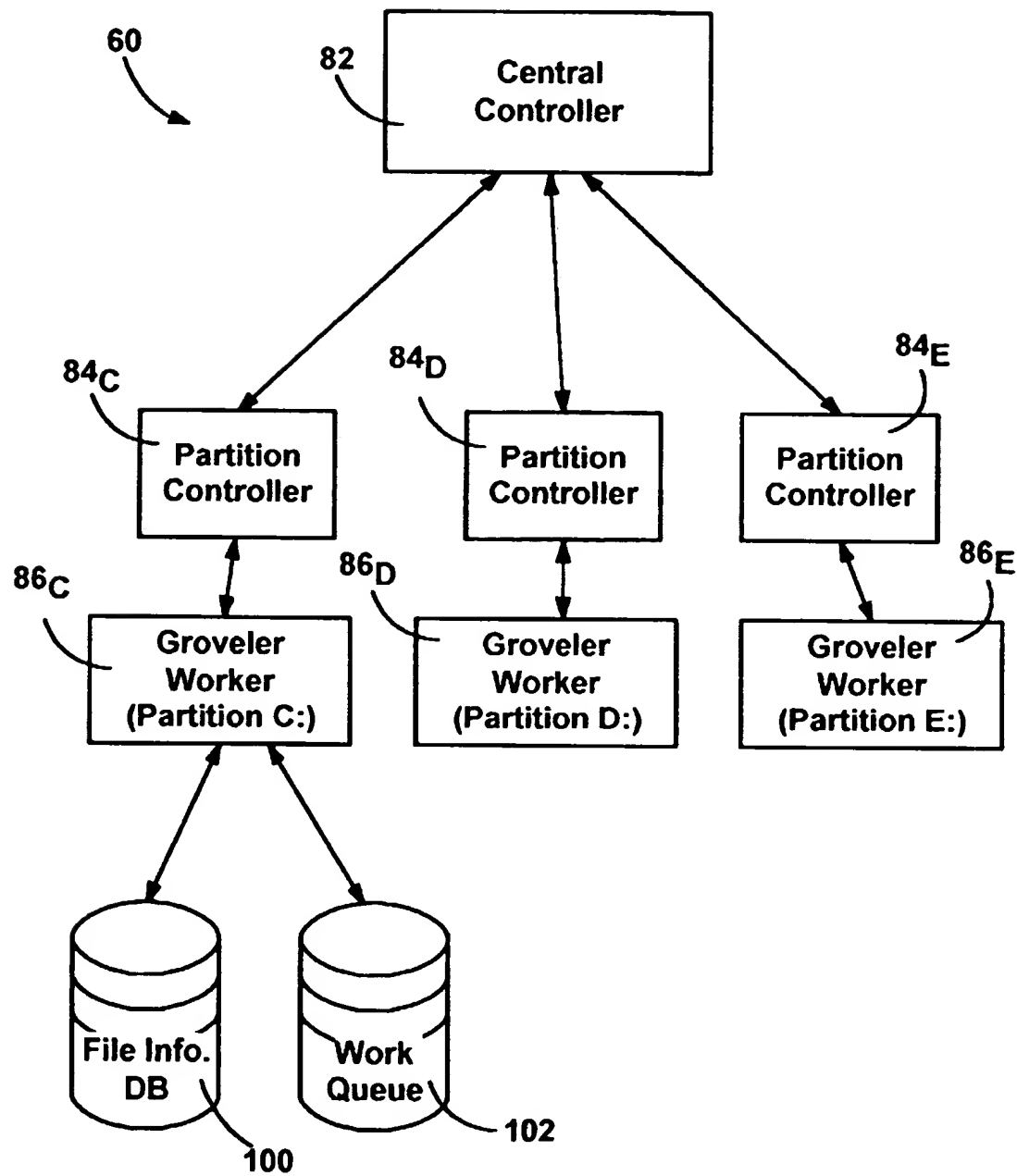


FIG. 2A



**FIG. 3**

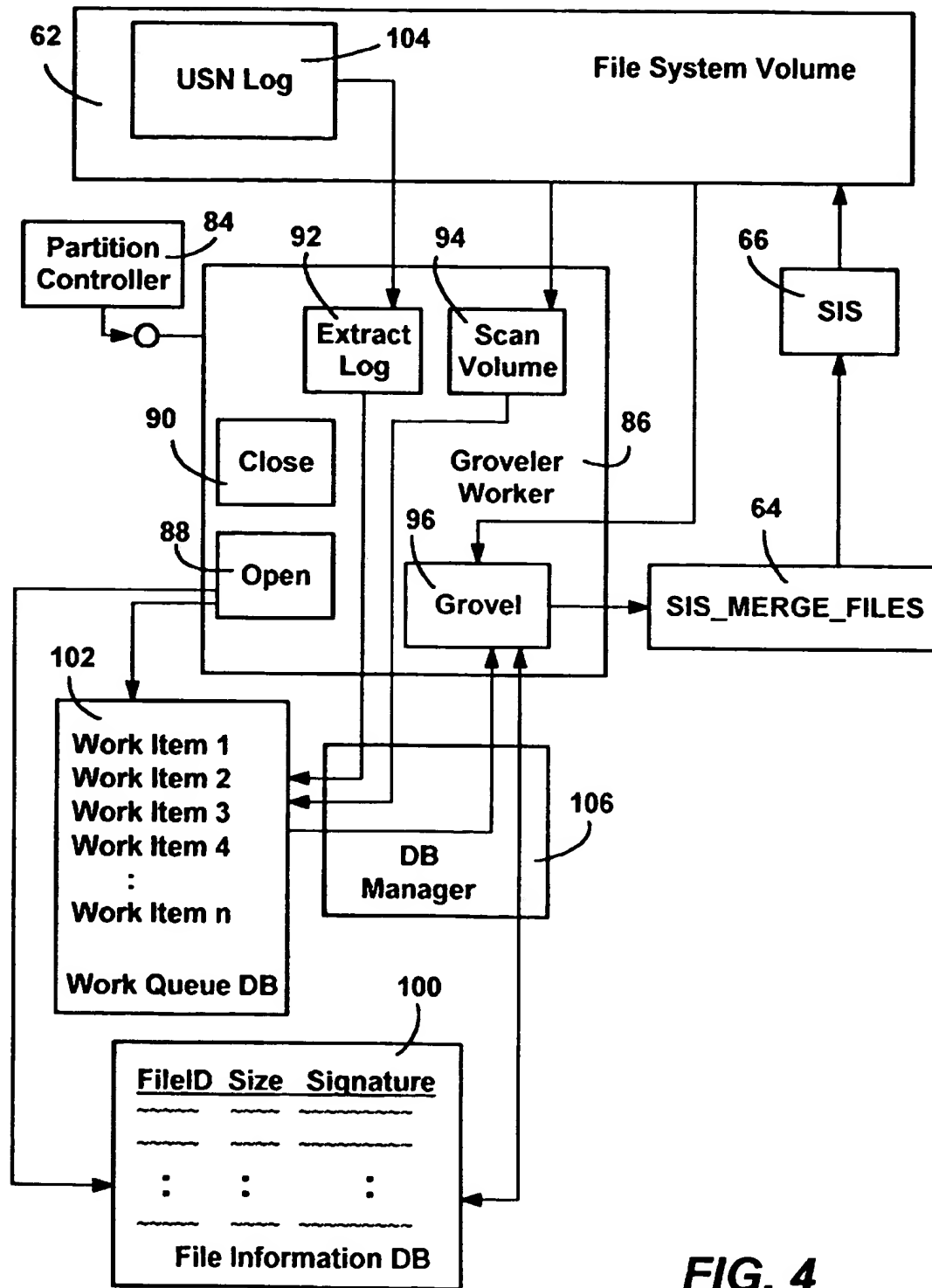
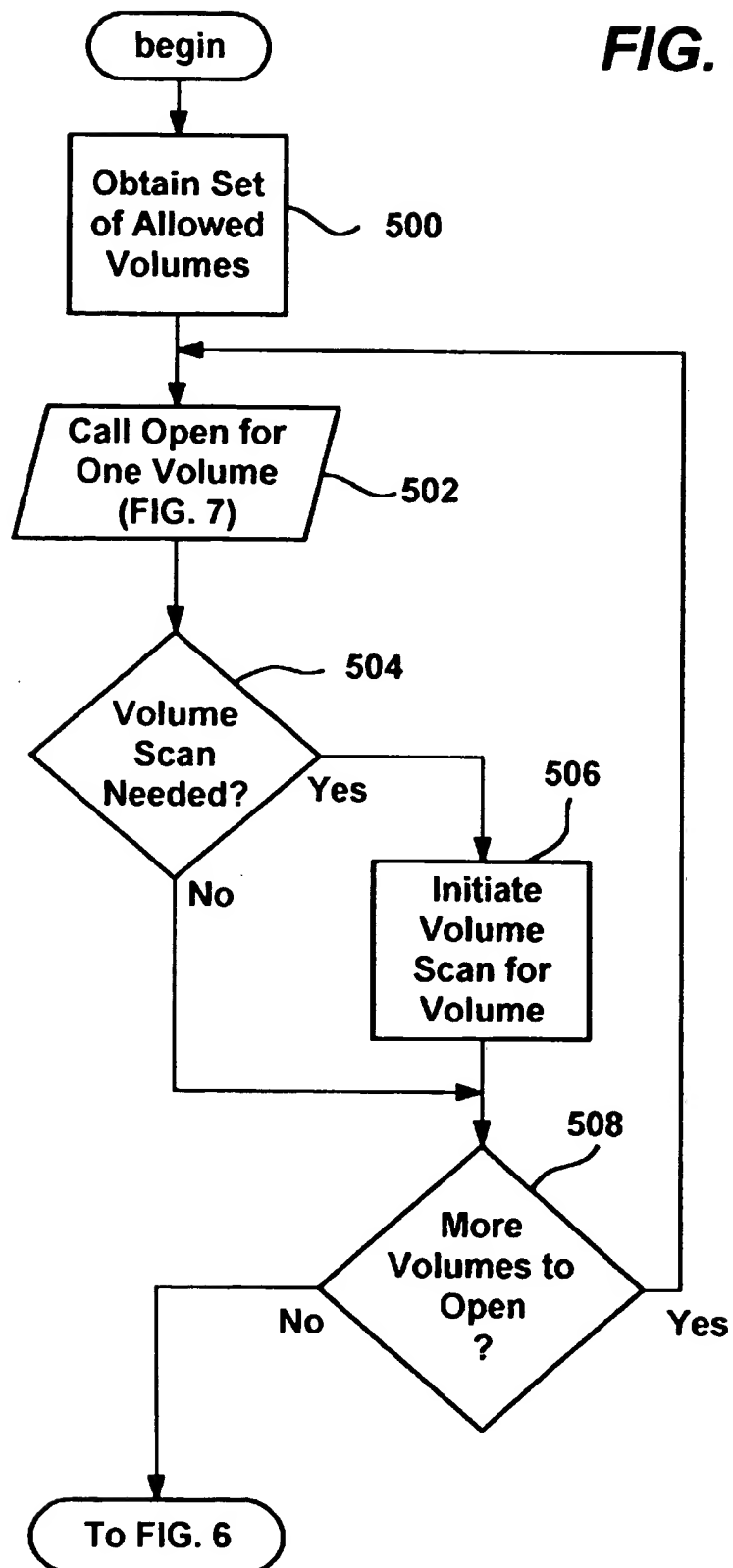


FIG. 4

FIG. 5

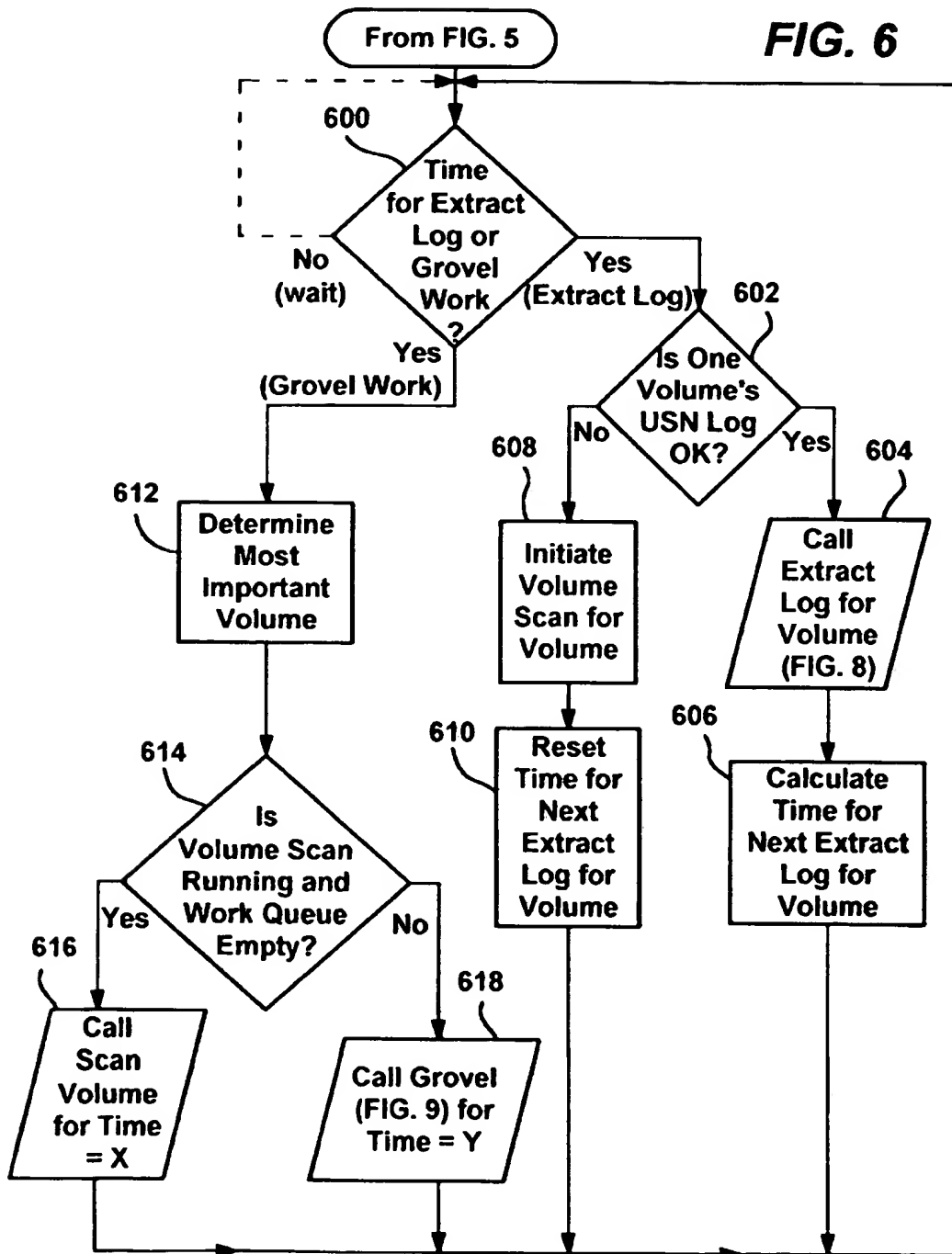


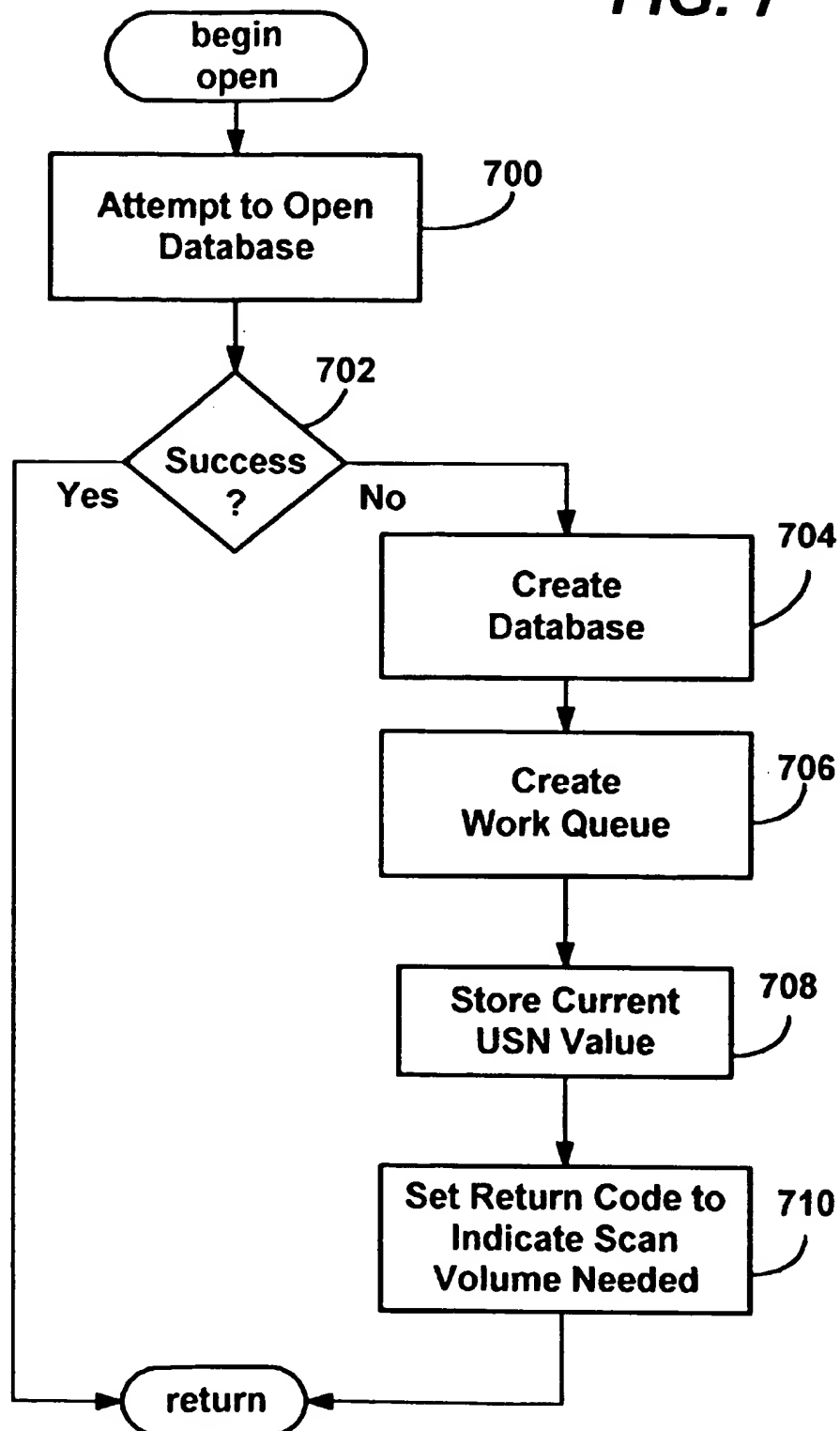
FIG. 7

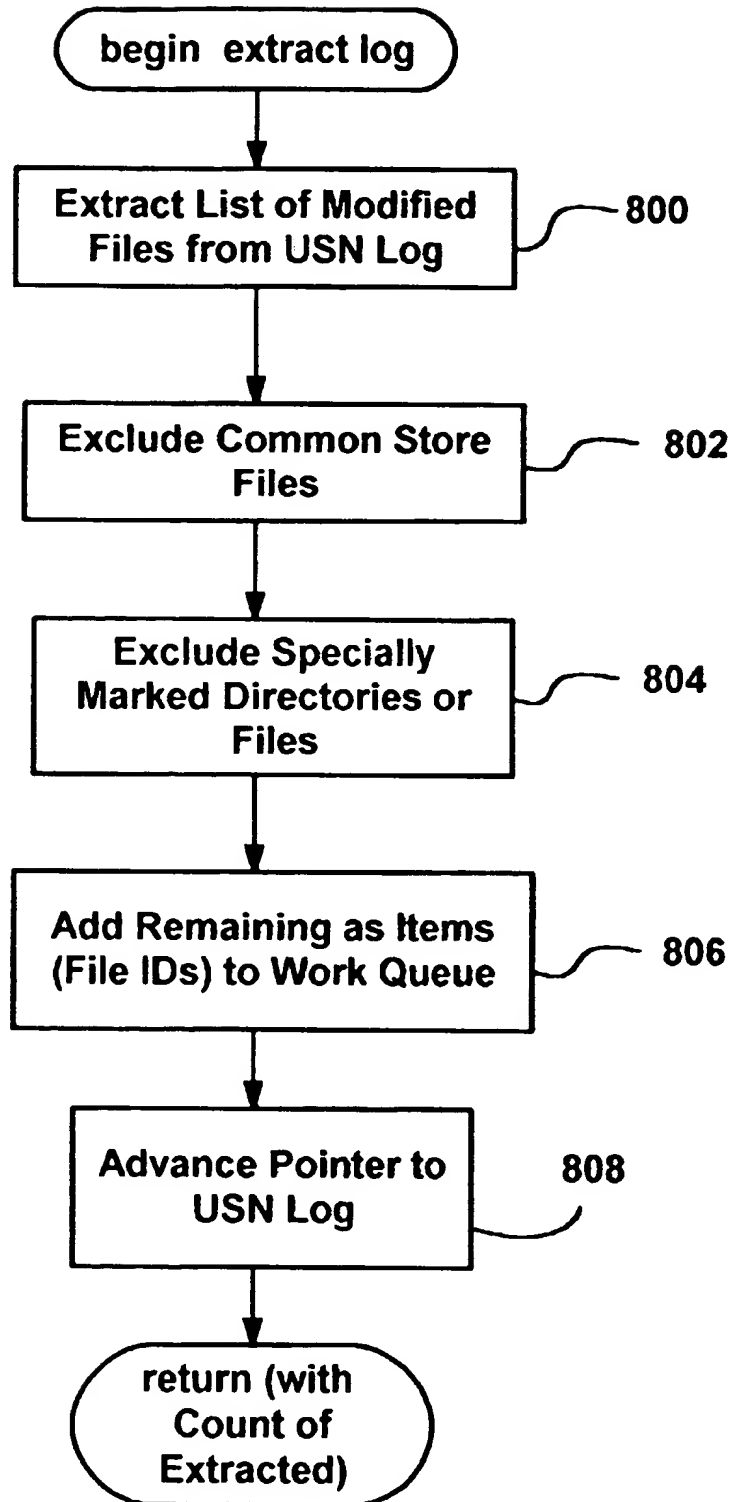
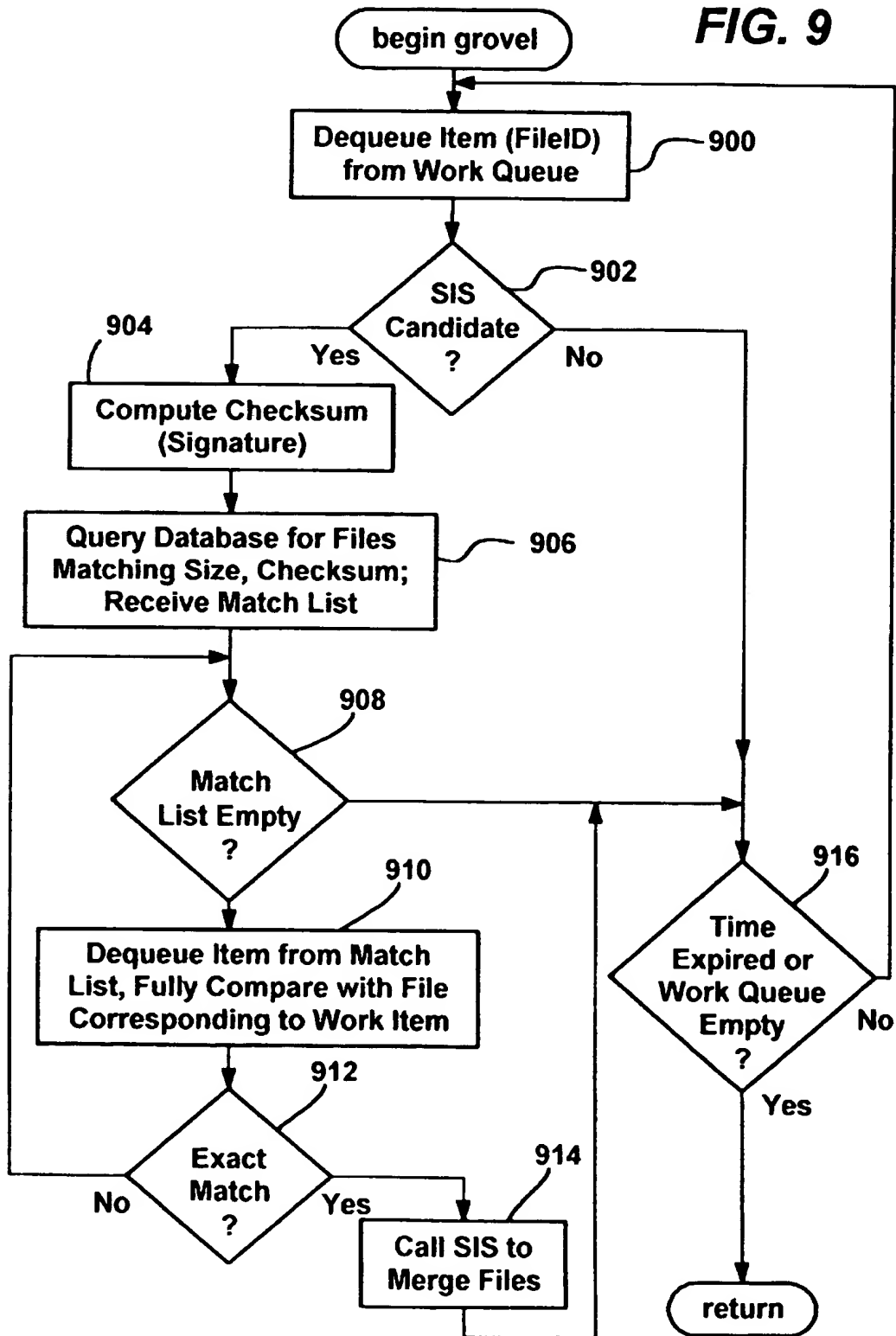
FIG. 8

FIG. 9

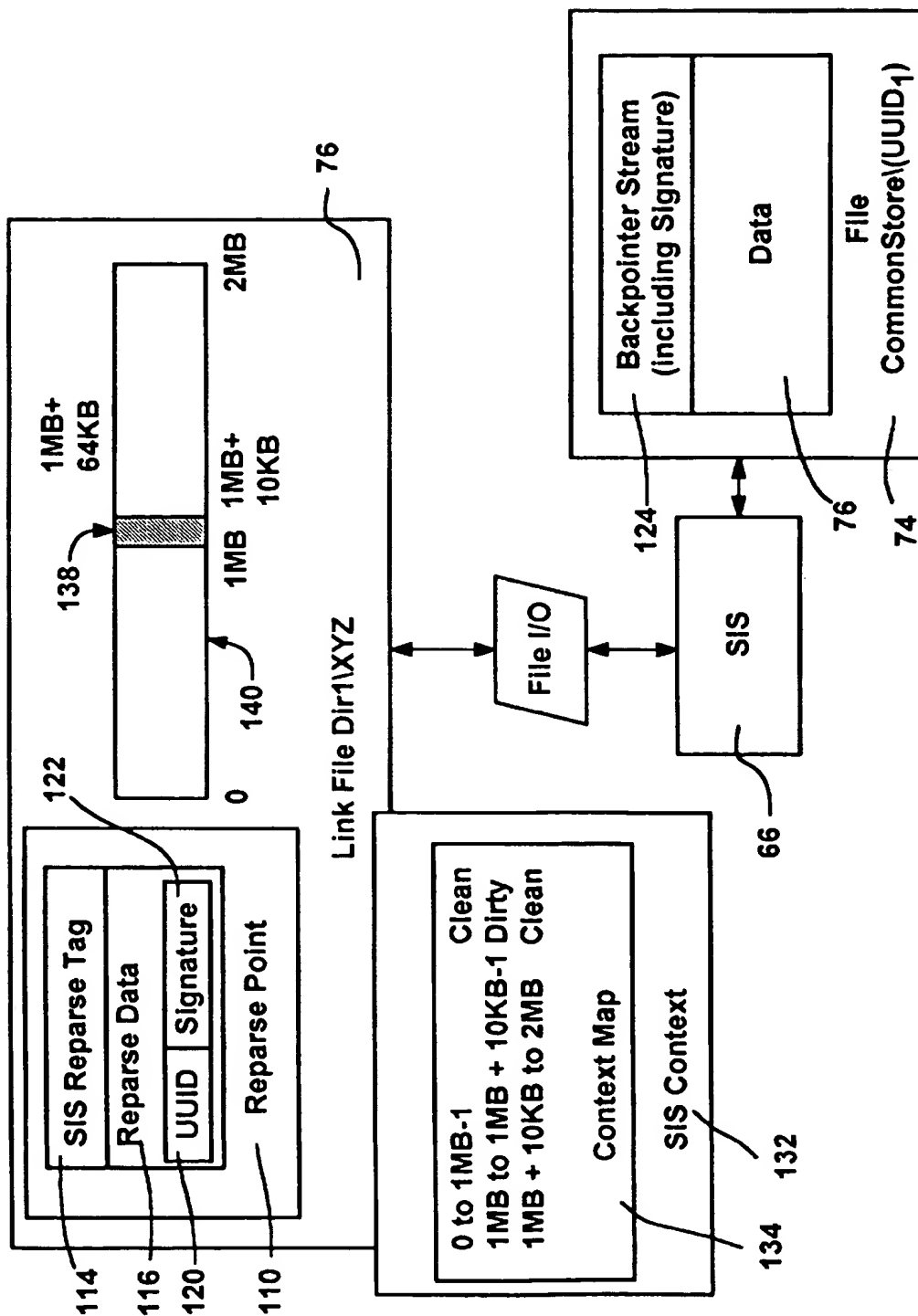


FIG. 10

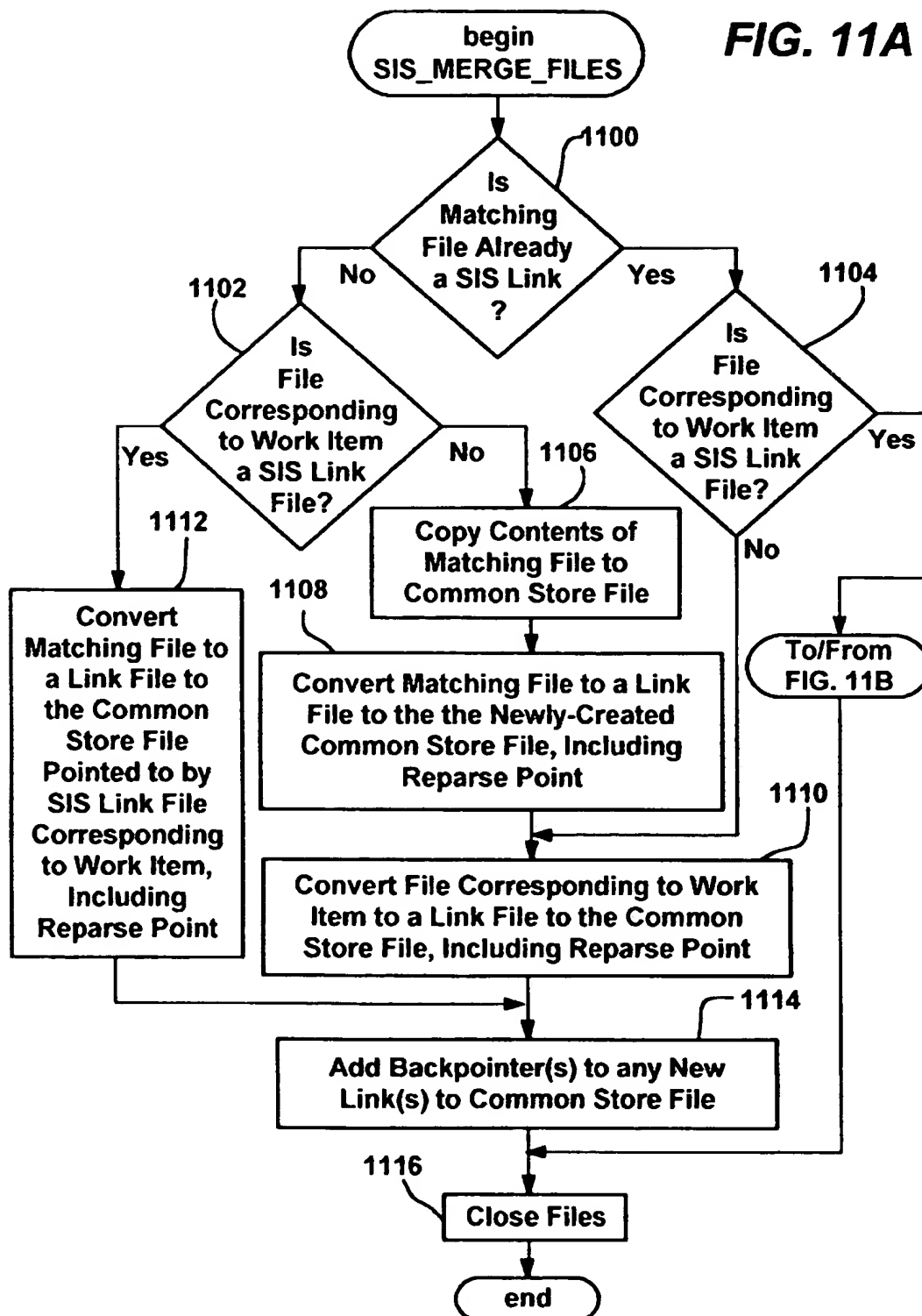
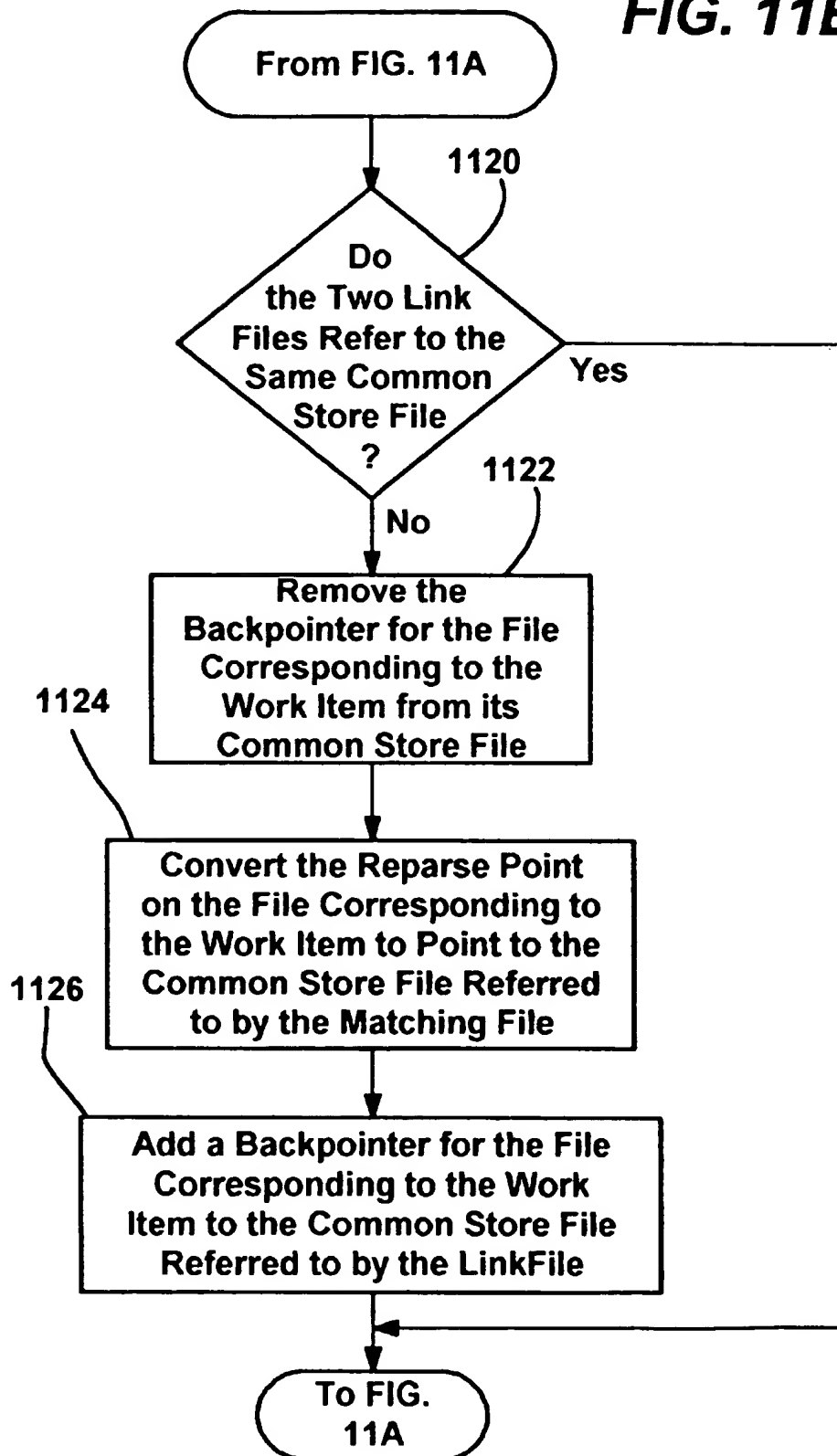
FIG. 11A

FIG. 11B

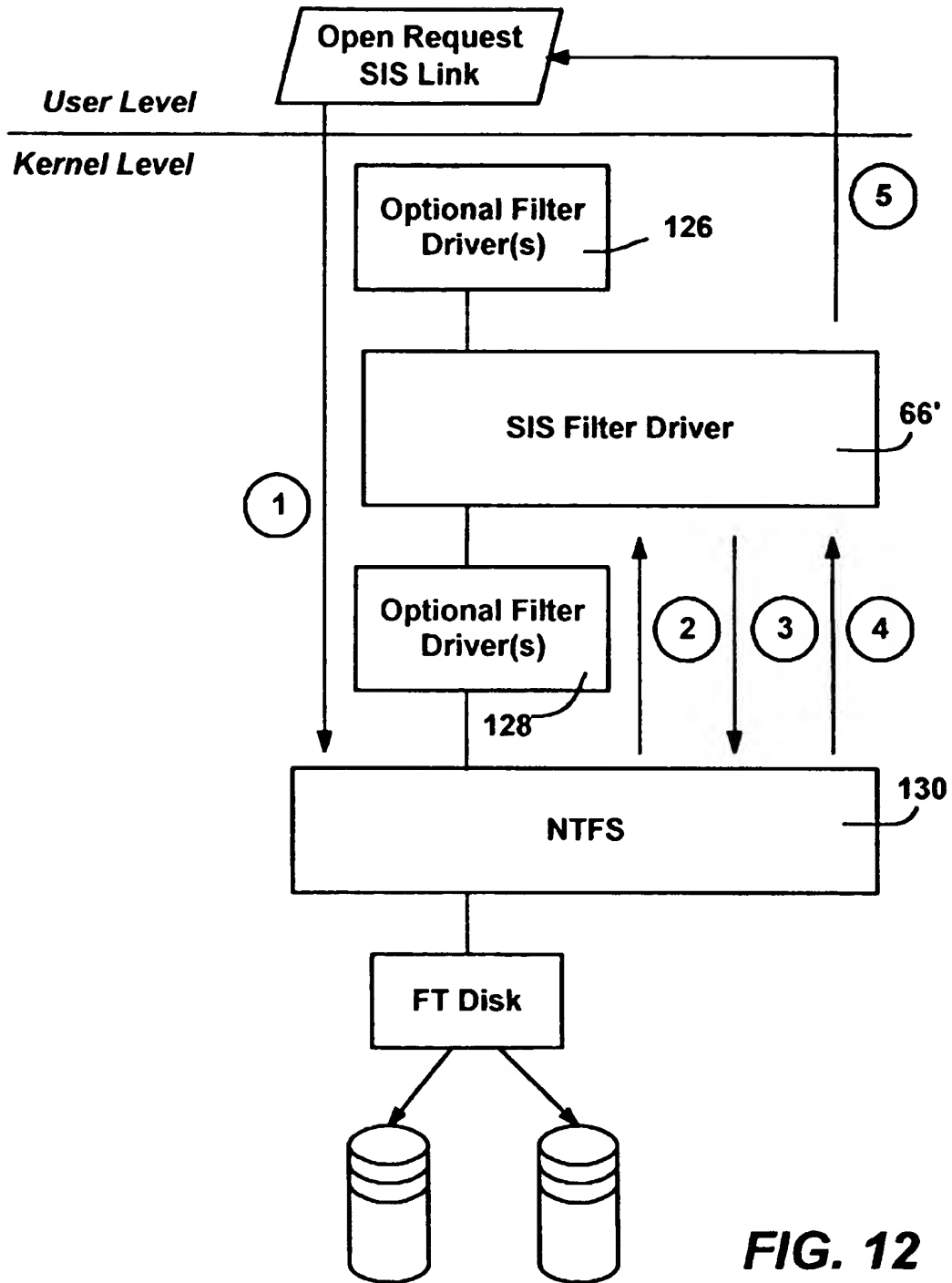
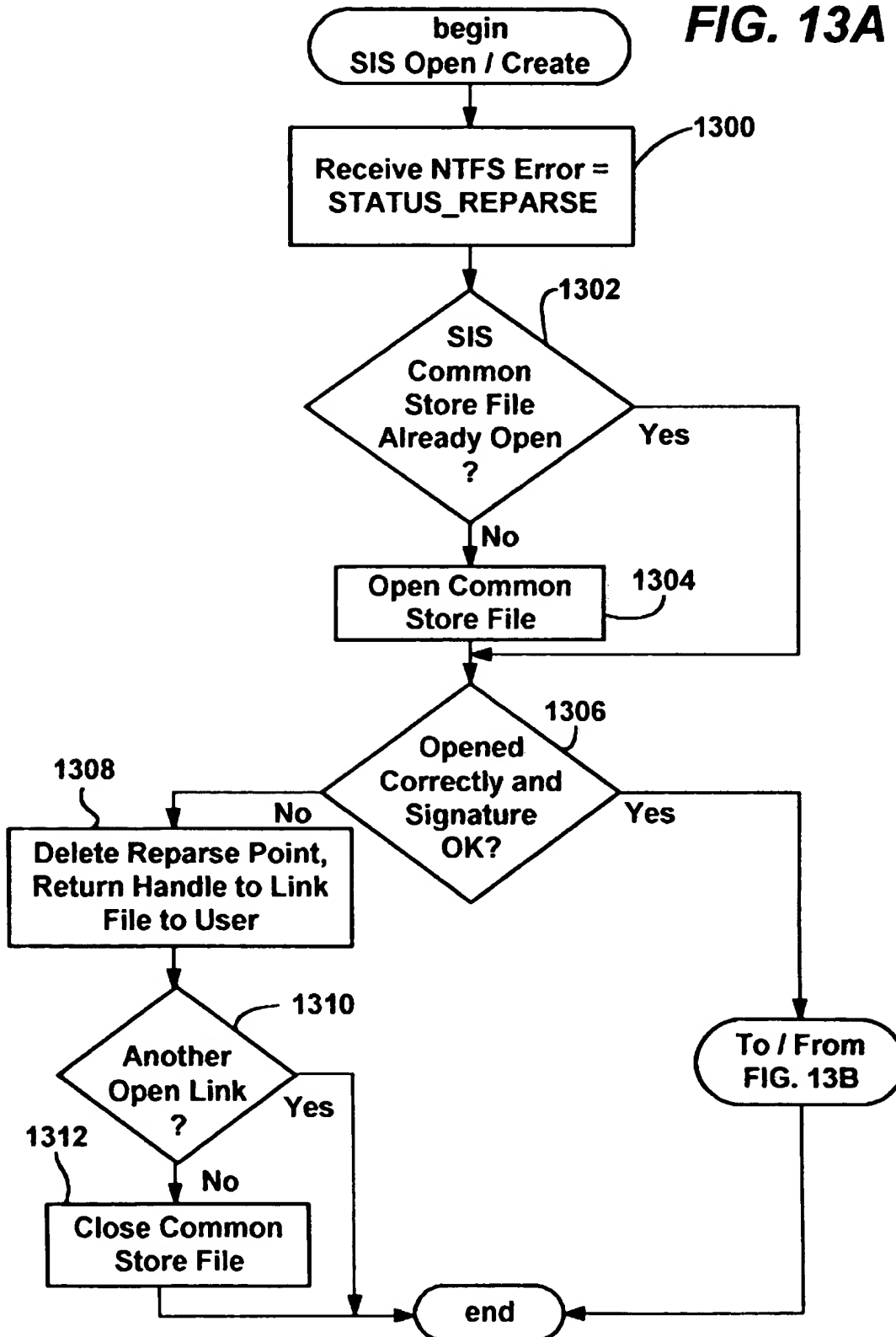
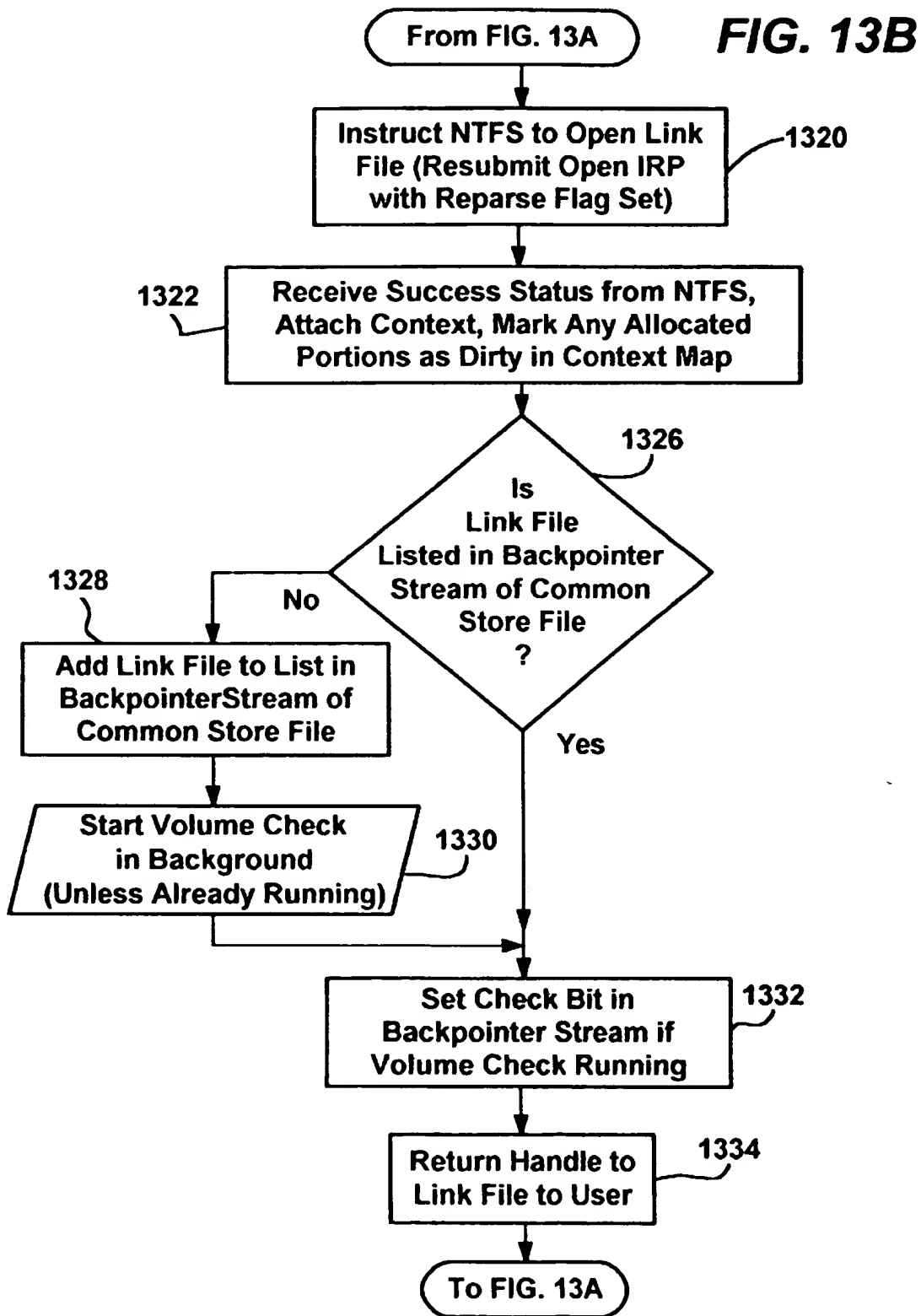


FIG. 12

FIG. 13A



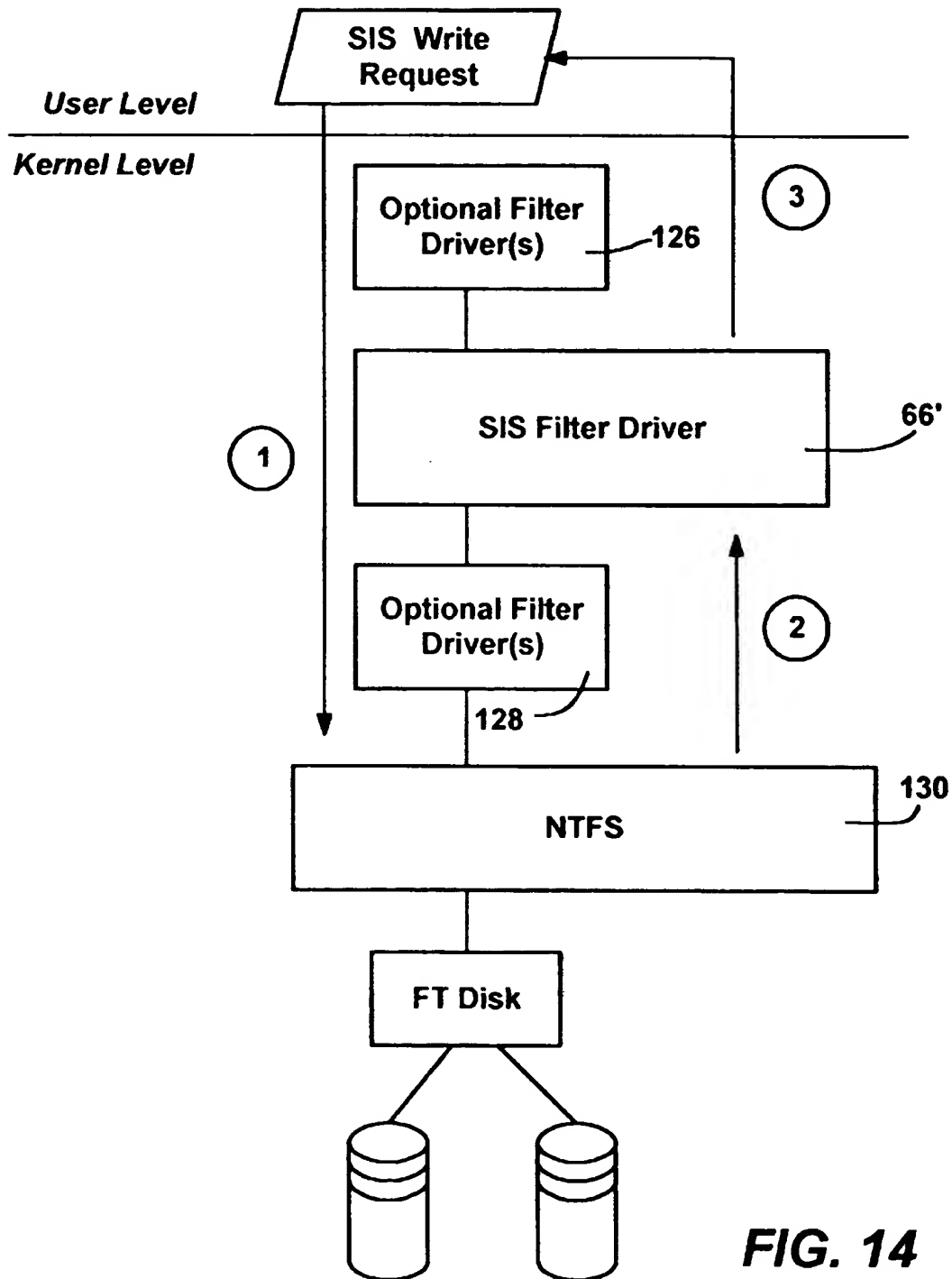
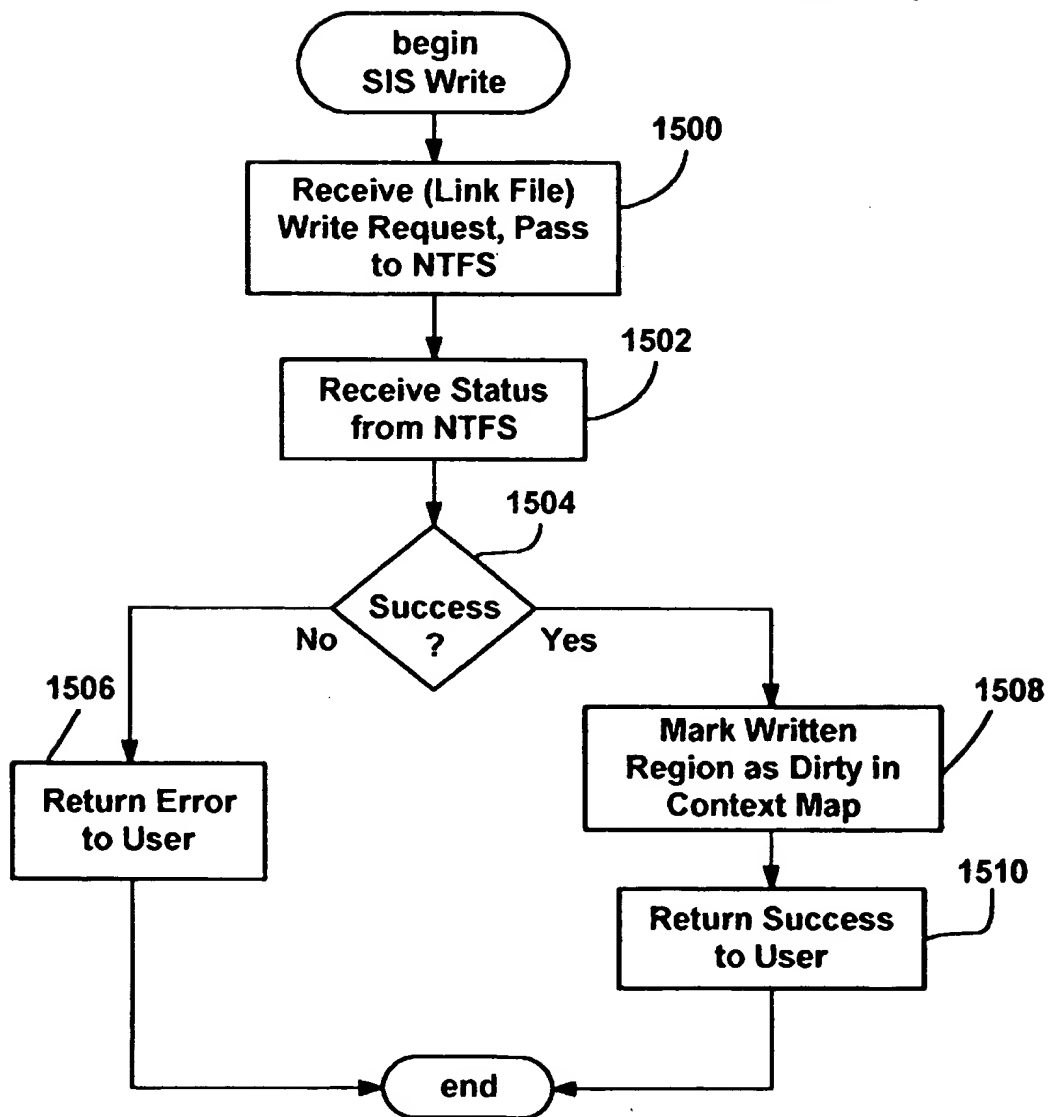


FIG. 14

FIG. 15

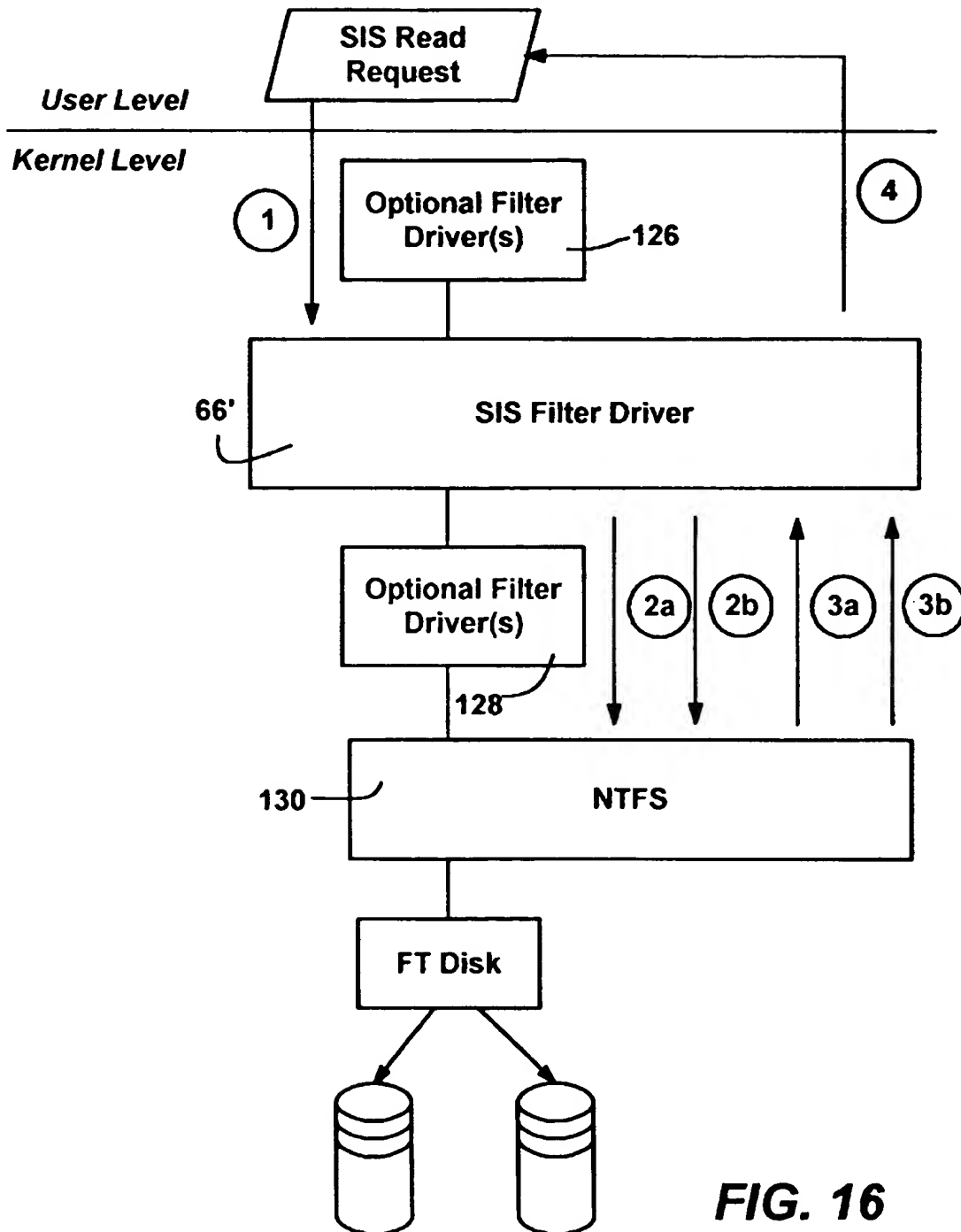
**FIG. 16**

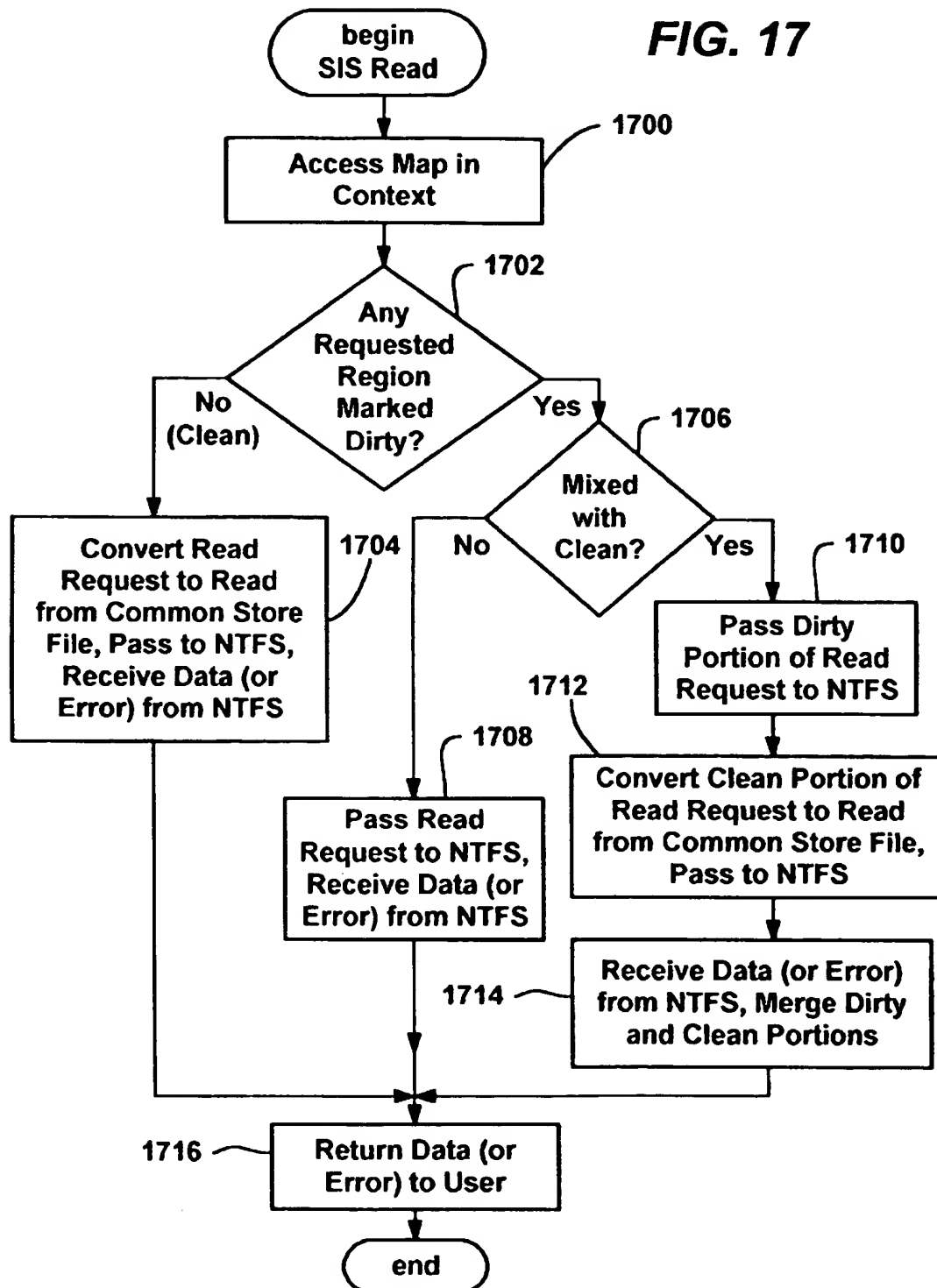
FIG. 17

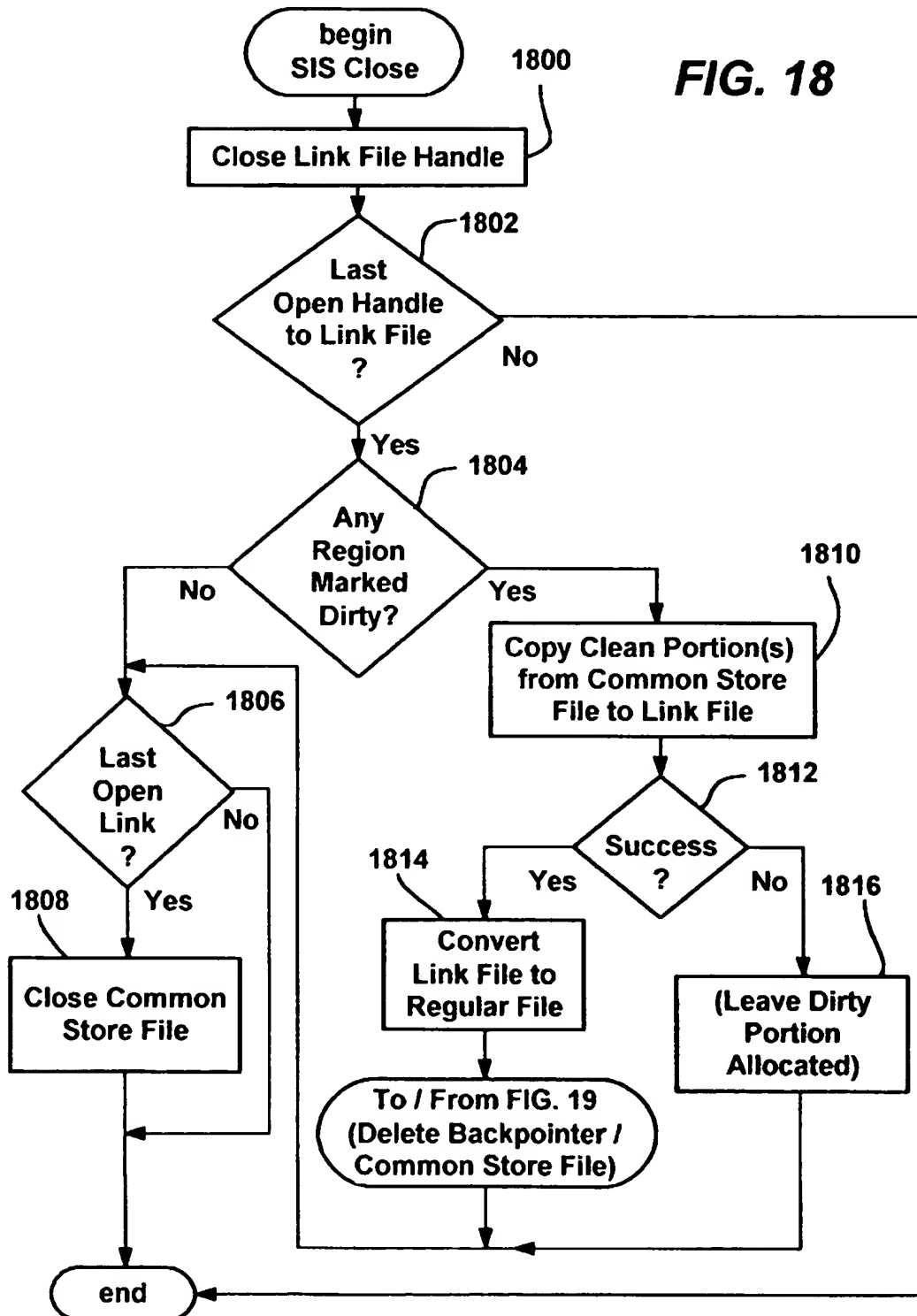
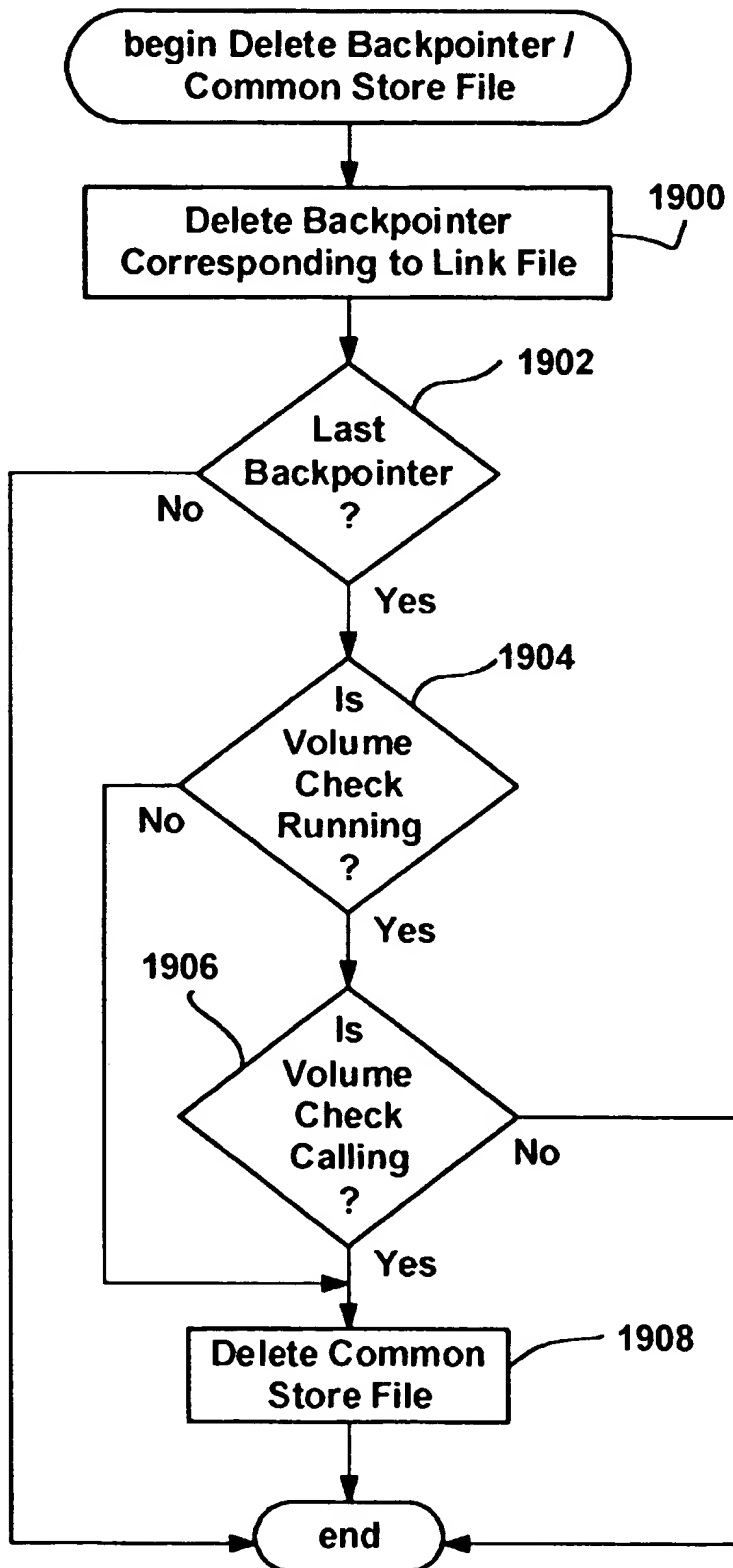
FIG. 18

FIG. 19

METHOD AND SYSTEM FOR AUTOMATICALLY MERGING FILES INTO A SINGLE INSTANCE STORE

TECHNICAL FIELD

The invention relates generally to computer systems and data storage, and more particularly to identifying and merging files of a file system having common properties.

BACKGROUND OF THE INVENTION

The contents of a file of a file system may be identical to the contents stored in one or more other files. While some file duplication tends to occur on even an individual user's personal computer, duplication is particularly prevalent on networks set up with a server that centrally stores the contents of multiple personal computers. For example, with a remote boot facility on a computer network, each user boots from that user's private directory on a file server. Each private directory thus ordinarily includes a number of files that are identical to files on other users' directories. As can be readily appreciated, storing the private directories on traditional file systems consumes a great deal of disk and server file buffer cache space.

Techniques that have been used to reduce the amount of used storage space include linked-file or shared memory techniques, essentially storing the data only once. However, when these techniques are used in a file system, the files are not treated as logically separate files. For example, if one user makes a change to a linked-file, or if the contents of the shared memory change, every other user linked to that file sees the change. This is a significant drawback in a dynamic environment where files do change, even if not very frequently. For example, in many enterprises, different users need to maintain different versions of files at different times, including traditionally read-only files such as applications. As a result, linked-file techniques would work well for files that are strictly read-only, but these techniques fail to provide the flexibility needed in a dynamic environment.

Another problem with these techniques is that identifying identical files becomes a complex task as the number of files on a file system volume increases. For example, a disk drive may store thousands of files, and each time a new file is written to a disk or a file is changed, a potential for file duplication exists. At times a user may know when files are duplicates of one another, and thus can manually request that the file data be shared, however relying on a user to detect such conditions is unpredictable, and for large numbers of files, inefficient and/or impractical. One possible solution is to run a utility at system start-up that scans a file system's files for duplicates, however this solution becomes unacceptably slow even with only a few thousand documents. Moreover, such a solution would not work well for users who seldom reboot a machine. Indeed, as more and more disk space is consumed, sharing files becomes a more valuable tool for preserving disk space, and thus a real-time solution could reclaim space when most needed. However, scanning even a relatively modest number of files in a file system volume for one or more duplicates, such as each time that a file is closed, consumes a great deal of time and machine resources, and thus is also impractical.

SUMMARY OF THE INVENTION

Briefly, the present invention provides a method and system for automatically identifying common files of a file system and merging those files into a single instance of the

data, having one or more logically separate links thereto representing the original files. A groveler facility maintains a database of information about the files on a volume of a file system, the information for each file including a file size and checksum (signature) based on the file contents. The groveler includes a component that periodically acts in the background to scan the USN log, a log that dynamically records file system activity, whereby new or modified files detected in the USN log are queued as work items, each work item representing a file. The entire volume also may be scanned to add work items to the queue, which takes place initially when the queue is created, or when there is a potential problem with the USN log.

The groveler includes another component that periodically removes items from the queue, calculates the signature of the corresponding file contents, and uses the signature and file size to query the database for matching files. The groveler component then compares any matching files with the file corresponding to the work item for an exact duplicate, and if found, calls a single instance store facility to merge the files and create independent links to those files.

Other advantages will become apparent from the following detailed description when taken in conjunction with the drawings, in which:

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram representing a computer system into which the present invention may be incorporated;

FIGS. 2A-2B are block diagrams representing various components for working with single instance store (SIS) link files and SIS common store files, including the automatic identifying and merging of duplicate files in accordance with an aspect of the present invention;

FIG. 3 is block diagram representing various components of a groveler for automatically identifying duplicate files for merging, in accordance with an aspect of the present invention;

FIG. 4 is block diagram representing various components connected to a groveler worker object of FIG. 3;

FIGS. 5 and 6 comprise a flow diagram generally representing the steps taken to call functions of a groveler worker to automatically identify and merge duplicate files in accordance with one aspect of the present invention;

FIG. 7 is a flow diagram generally representing the steps taken by the groveler worker open function;

FIG. 8 is a flow diagram generally representing the steps taken by the groveler worker extract log function;

FIG. 9 is a flow diagram generally representing the steps taken by the groveler worker grovel function;

FIG. 10 is block diagram representing various components of a SIS link file and SIS common store file;

FIGS. 11A-11B comprise a flow diagram generally representing the steps taken to merge duplicate files into a SIS common store file;

FIG. 12 is a representation of a SIS link file open request passing through a preferred SIS and file system architecture;

FIGS. 13A and 13B comprise a flow diagram generally representing the steps taken by the SIS facility to handle the open request represented in FIG. 12;

FIG. 14 is a representation of a SIS link file write request passing through a preferred SIS facility;

FIG. 15 is a flow diagram generally representing the steps taken by the SIS facility to handle the write request represented in FIG. 14;

3

FIG. 16 is a representation of a SIS link file read request passing through a preferred SIS facility;

FIG. 17 is a flow diagram generally representing the steps taken by the SIS facility to handle the read request represented in FIG. 16;

FIG. 18 is a flow diagram generally representing the steps taken by the SIS facility to handle a SIS link file close request; and

FIG. 19 is a flow diagram generally representing the steps taken by the SIS facility to handle a SIS link file delete request.

DETAILED DESCRIPTION OF THE INVENTION

Exemplary Operating Environment

FIG. 1 and the following discussion are intended to provide a brief general description of a suitable computing environment in which the invention may be implemented. Although not required, the invention will be described in the general context of computer-executable instructions, such as program modules, being executed by a personal computer. Generally, program modules include routines, programs, objects, components, data structures and the like that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multi-processor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

With reference to FIG. 1, an exemplary system for implementing the invention includes a general purpose computing device in the form of a conventional personal computer 20 or the like, including a processing unit 21, a system memory 22, and a system bus 23 that couples various system components including the system memory to the processing unit 21. The system bus 23 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory includes read-only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system 26 (BIOS), containing the basic routines that help to transfer information between elements within the personal computer 20, such as during start-up, is stored in ROM 24. The personal computer 20 may further include a hard disk drive 27 for reading from and writing to a hard disk, not shown, a magnetic disk drive 28 for reading from or writing to a removable magnetic disk 29, and an optical disk drive 30 for reading from or writing to a removable optical disk 31 such as a CD-ROM, DVD-ROM or other optical media. The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical drive interface 34, respectively. The drives and their associated computer-readable media provide non-volatile storage of computer readable instructions, data structures, program modules and other data for the personal computer 20. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 29 and a removable optical disk 31, it should be appreciated by those skilled in the art that other types of computer readable media that can store

4

data that are accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read-only memories (ROMs) and the like may also be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24 or RAM 25, including an operating system 35 (preferably Windows® 2000). The computer 20 includes a file system 36 associated with or included within the operating system 35, such as the Windows NT® File System (NTFS), one or more application programs 37, other program modules 38 and program data 39. A user may enter commands and information into the personal computer 20 through input devices such as a keyboard 40 and pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor 47, personal computers typically include other peripheral output devices (not shown), such as speakers and printers.

The personal computer 20 may operate in a networked environment using logical connections to one or more remote computers 49. The remote computer (or computers) 49 may be another personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the personal computer 20, although only a memory storage device 50 has been illustrated in FIG. 1. The logical connections depicted in FIG. 1 include a local area network (LAN) 51 and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, Intranets and the Internet.

When used in a LAN networking environment, the personal computer 20 is connected to the local network 51 through a network interface or adapter 53. When used in a WAN networking environment, the personal computer 20 typically includes a modem 54 or other means for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

The present invention is described herein with reference to Microsoft Corporation's Windows 2000 (formerly Windows NT®) operating system, and in particular to the Windows NT® file system (NTFS). Notwithstanding, there is no intention to limit the present invention to Windows® 2000, Windows NT® or NTFS, but on the contrary, the present invention is intended to operate with and provide benefits with any operating system, architecture and/or file system.

The Groveler

Turning now to FIGS. 2A-2B, there is shown a general concept of a groveler 60 and a single instance store (SIS) facility and architecture underlying a preferred implementation of the present invention, which may be implemented

5

in the computer system 20. In accordance with one aspect of the present invention and as described in detail below, as represented in FIG. 2A, in general, the groveler 60 finds files having duplicate data in a file system volume 62. Via a file system control named SIS_MERGE_FILES 64, the groveler 60 calls the Single Instance Store (SIS) facility 66 to merge the duplicate files into a single instance of data with links thereto. The SIS_MERGE_FILES control 64 may be implemented as a Windows 2000 file system control, recognized by the SIS facility 66. One such (SIS) facility 66 is described below, and is further described in copending United States Patent Application entitled "Single Instance Store for File Systems," assigned to the assignee of the present invention, filed concurrently herewith, and hereby incorporated by reference herein in its entirety. Note that alternatively, a user, via a SIS_COPYFILE request 68 to the SIS facility 66, may explicitly (manually) request that a source file be copied to a destination file as a SIS copy of the file.

As shown in FIGS. 2A and 2B, the groveler 60 finds, for example, that files 70, 72 (named Dir1\XYZ and Dir2\ABC) have duplicate data. Note that the files may be in separate directories of the volume 62 or in the same directory. When such duplicate files 70, 72 are identified, the groveler 60 calls the SIS facility 66 via the SIS_MERGE_FILES control request 64. As described below and as generally shown in FIG. 2B, the call to the SIS facility 66 normally results in a single instance representation 74 of the original files 70, 72 with links 76, 78 thereto, each link corresponding to one of the original files, e.g., the user sees each link file as if it was the original file. The common store file 74 is maintained in a common store directory 80 of such files.

Each SIS link file 76, 78 is a user file that is managed by the SIS facility 66, while the common store 80 is preferably a file system directory that is not intended to be visible or accessible to users. Note that the single instance representation 74 need not actually be a file system file in a file system directory, but may be stored in some other data structure. Thus, as used herein, the link file, common store file and/or single instance file components are intended to comprise any appropriate data structure that can hold at least part of a file's contents. Notwithstanding, the link files 76, 78 may be maintained on the same file system volume 62, as is the common store file 74 and the common store directory 80. This enables removable media to take the links and common store with it when removed, prevents formatting one volume (e.g., D:) from losing the common store file or links of another volume (e.g., C:), and so forth.

Repeating the SIS_MERGE_FILES (and/or SIS_COPYFILE) processes for any other files that have the same data will add links without substantially adding to the single instance of the file. In this manner, for example, an administrator user of a file server may place the links for many client users on each user's private directory, while maintaining only one instance of the file on the server. Note that it is feasible to have a SIS common store file with only one link thereto, while alternatively, a control may be implemented that allows more than two files to be specified at the same time for merging into a single instance representation thereof. As also described below, it also may occur that the groveler 60 detects a file (that is not a SIS link file) but already has a single instance representation of its data in the common store directory 80. In such an instance, the non-SIS link file may be converted (as described below) to a link to the existing single instance file.

In accordance with one aspect of the present invention and as generally represented in FIG. 3, to accomplish

6

detection of duplicate files for merging purposes, the groveler 60 includes a central controller 82, preferably implemented as an instantiated object (e.g., a C++ object) with one or more defined interfaces thereto. The central controller 82 regulates the operation of one or more partition controllers 84_C-84_E, (three are shown in FIG. 4), one partition controller per file system volume. In turn, the partition controllers 84_C-84_E each have a groveler worker 86_C-86_E associated therewith that when activated, individually attempt to identify duplicate files in their corresponding file system volume.

The central controller 82 synchronizes the operation of the partition controllers 84_C-84_E across multiple volumes, for example, such that only one runs at a time depending on available system resources. In turn, when allowed to operate, each partition controller 84_C-84_E calls functions of its corresponding groveler worker 86_C-86_E.

As represented in FIG. 4, each groveler worker 86 is a single process, and includes an open function 88, close function 90, extract log function 92, scan volume function 94 and grovel function 96. In general, calling the open function 88 causes the groveler worker 86 to open (or create if needed) a database 100 of file information and a queue 102 of work items, also conveniently stored as a database. The close function 90 takes care of any cleanup operations before the groveler worker is shut down.

The extract log function 92 uses a USN (Update Sequence Number) log 104 to add items (file identifiers) to the work item queue 102. As is known, the USN log 104 is a function of the underlying NTFS filesystem 130 that dynamically records changes to a file system volume's files by storing change information indexed by a monotonically increasing sequence number, the USN. The extract log function 92 reads records from the USN log 104, each time starting from where it previously left off (as tracked by recording the USN), filters out those records that do not deal with new or modified files, and adds items to the work item queue 102 that correspond to new or modified files.

Calling the scan volume function 94 places work items (file identifiers corresponding to files in the volume) into the work item queue 102 via a depth first search of the file system volume 62. The scan volume function 94 is time limited, whereby when called, it places as many files as possible into the work item queue 102 within its allotted time, and resumes adding files from where it left off when called again. Note that the scan volume function 94 may be given some filtering capabilities, e.g., such that it will not add common store files to the work item queue 102, however at present the scan volume function 94 merely adds file identifiers as items to the work item queue 102 and any filtering is performed when work items are dequeued. The scan volume function 94 is called only when needed, e.g., when the work item queue 102 is first created, or if a problem occurs with the USN log 104 or the database 100, since the extract log function 92 may not be provided with the proper file change information.

The grovel function 96 removes items from the work item queue 102, and processes each removed item to determine if it meets some criteria, e.g., whether the file corresponding to that work item has a duplicate file in the volume 62. To this end, the grovel function 96 computes a checksum (signature) from the file's data, and queries the file information database 100 via a database manager 106 (including a query engine) to see if one or more files in the volume have the same checksum and file size. At the same time, the database manager 106 updates the file information database 100 as needed with the file information, e.g., adds new records or changes existing records by storing or modifying

the file size and signature indexed by the file ID in the database 100. If at least one matching file is found, the groveler function 96 performs a byte-by-byte comparison with the matching file set to determine if one of the files is an exact duplicate, and if exact, calls the SIS facility 66 via the SIS_MERGE_FILES control 64 to merge the files. In this manner, duplicate files are automatically identified and combined in a rapid manner.

Turning to an explanation of the present invention with particular reference to the flow diagrams of FIGS. 5-9, the groveler 60 begins at step 500 when the central controller 82 determines which of the volumes are allowed to be groveled. A preferred way to determine when and how to run a background process (such as the way in which the groveler 60 is typically run) uses a technology sometimes referred to as "back off technology," that measures the actual performance of background tasks, including the performance of input/output (I/O) operations. The measurements are used to statistically determine when the background process is likely degrading the performance of a foreground process, in which event the background process temporarily suspends its execution, i.e., backs off. As a result, certain volumes may be too busy to grovel, for example, or locked by a disk utility, whereby only a subset of the total number of volumes may be groveled at a given time. Such back off technology is described in copending United States Patent Application entitled "Method and System for Regulating Background Tasks Using Performance Measurements," filed concurrently herewith, assigned to the assignee of the present invention and hereby incorporated by reference herein in its entirety. Notwithstanding, virtually any mechanism including CPU scheduling priority may be used to determine when to execute the groveler 60 on a volume, and moreover, the groveler 60 may be periodically run as a foreground process.

Once the set of volumes that can be groveled is determined at step 500, the open function of the groveler worker (e.g. object 86) is called at step 502 for one of the volumes, and as represented by step 508, is repeated for each volume. Note that although not shown, the groveler worker (object) 86 is instantiated (if not already instantiated) before calling its functions. For each volume, if at step 504 the result of the open call indicates a volume scan is needed, a scan is initiated at step 506 by setting a flag associated with the volume. After repeating this process via step 508 for each volume of the set that can be groveled, a main loop (FIG. 6) is entered, as described below.

FIG. 7 shows the general operation of the open function 88 for a given volume, beginning at step 700 where the groveler worker 86 attempts to open the file information database 100. If not successful as detected by step 702, the groveler worker 86 creates a new database 100 at step 704 for the volume along with a new work item queue 102 at step 706. The groveler worker 86 also stores the volume's current USN value at step 708, and sets a return code at step 710 to indicate that the partition controller 84 needs to call the scan volume function 94 to begin filling the work item queue 102 with the volume's file identifiers as described above. The current USN number is stored so that the extract log function 92 will be able to handle any file changes that happen after the volume scan is started.

FIG. 6 shows the main loop for operating the groveler functions. In each pass of the loop, for each volume it is determined at step 600 whether it is time for an extract log function call, time to grovel, or neither. If it is time for neither function, the process waits until the appropriate time is achieved, as represented by the dashed line "looping" back to step 600. Note that the time to grovel a volume is

regulated via the aforementioned back-off technology, and thus ordinarily varies with respect to the activity of foreground processes so as to limit interference of the groveling operations therewith. Note that back-off technology also may vary the frequency of groveling based on other factors, such as to grovel more frequently when disk space is deemed low in an increased-priority effort to gain free space by merging files.

The extract log function 92 is called at a frequency that varies in an attempt to place a constant number of items in the work item queue per call, and thus the frequency of calling the extract log function 92 is based on the amount of file system volume activity taking place. For example, if disk activity is at a high rate, a large number of USN records will be extracted from the USN log 104, whereby the extract log function 92 will likely add a larger number of items to the work item queue 102 relative to times of slow disk activity. By using the number of records extracted during the most recent extract log function call to determine the time duration before the next call, a high number of extracted records will cause a higher rate of calling extract log, while a lower number will cause a lower rate of calling. Over a period of time, the changes to the rate of calling roughly provide the desired number of items being placed in the work item queue 102 per call. Note that the rates may be adjusted gradually to smooth out any abrupt changes in disk activity. This is done to trade off the expense of draining the USN log too frequently against the possibility that the log will overflow and force a potentially very expensive scan volume.

If it is time for the extract log function for any volume (e.g., the volume 62), the USN log for that volume 62 is checked at step 602 to determine whether it is correct. Note that the USN log is typically of a fixed size, and so only keeps a fixed number of entries, corresponding to the most recent updates to the volume, i.e., older entries are discarded. If there are more volume updates than space in the USN log between times when the groveler worker 86 looks at the log, it will miss some volume updates, and the USN log will be deemed to be incorrect. The USN log will also be deemed to be incorrect if it is corrupted, for instance by a data error on the underlying disk.

If at step 602 the USN log 104 is not correct, step 602 branches to step 608 to initiate a scan volume operation for that volume as described above. Step 610 resets the time for performing the next extract log function, as also described above.

If it is time to call the extract log function (step 600) and the USN log 104 is correct (step 602), the extract log function 92 is called at step 604 (as described via FIG. 8) to place items in the work item queue 102 corresponding to the USN entries since the last recorded USN record that was processed.

As represented in FIG. 8, the extract log function 92 begins by extracting a list of file identifiers corresponding to modified files from the USN log 104. Note that the application programming interface (API) or the like that allows the USN log 104 to be read may filter its return such that only selected types of files are returned, i.e., only those that are new or changed, and such that the same file is not listed multiple times. In addition, at step 802 the extract log function 92 may exclude certain files from the items to queue, such as common store files. For example, if a common store file is created from two identical files found by the groveler 60, appropriate entries will go into the USN log 104 to reflect this file system activity. These entries are flagged by the groveler 60 so that they are recognized and

not again processed by the groveler 60. Alternatively, these items may be filtered out by the USN retrieval process (API). Also, it is possible that the user or system may choose to exclude certain files (e.g., on a per file or per directory basis) from automatic merging, essentially overriding the groveler 60 for selected files. If any such files are identified, these files are excluded at step 804.

At step 806, the remaining files are added as items to the work queue, identified by their volume-unique file identifier. At step 808, the extract log function 92 records the last USN handled so that the extract log function 92 will begin at the correct location the next time it is called. As described above, the extract log function 92 returns a count of the USN entries extracted, from which the partition controller 84 calculates (at step 606 of FIG. 6) the next time to call the extract log function 92. Steps 806 and 808 are handled as an atomic database transaction so that a system crash in the middle will not result in the pointer being updated without the items being added to the work queue.

Returning to FIG. 6, following the extract log call, the time for the next extract log is calculated at step 606, and the process loops back to step 600. After repeating the process until it is not the extract time for any volume, step 600 branches to step 612 where the central controller 82 and/or partition controllers (e.g., 84_C-84_E) may determine which of the volumes is the most important to grovel relative to others. One criterion for determining relative importance includes how much free space is left on a volume. Additional criteria may include when the volume was last groveled and/or the results of that grovel operation, so that, for example, the same nearly-full volume is not always considered the most important, particularly if there are no files to merge thereon.

Once the most important volume is selected, if at step 614 a volume scan is in progress (has been initiated and not completed) and the work queue is empty, the scan volume function 94 of the groveler worker 86 is called to begin/continue a scan operation of the entire volume 62, i.e., to begin or continue filling the work item queue 102 with the volume's file identifiers as described above. The current USN number is stored so that the extract log function 92 will be able to handle any file changes that happen after the volume scan is started. As mentioned above, the scan volume operation is time limited, presently 200 milliseconds, although of course this time may be made variable. For example, the number of items actually queued per call may be returned by the scan volume function and used to determine the time duration for the next call.

If at step 614 a volume scan is not in progress, or if the work queue is not empty, the grovel function (FIG. 9) is called at step 618, after which the process returns to the main loop at step 600.

In the grovel function, as represented beginning at step 900, the first item is dequeued from the work item queue 102. More particularly, the work item queue 102 is conveniently maintained as a database and accessed by the database manager 106 using transactions, whereby each item is atomically handled or not handled. As a result, the work item is not considered removed until fully processed, whereby if a system failure (crash) occurs before complete processing, the work item is not lost. However, for purposes of simplicity herein, the work item may be considered dequeued at this time, and step 902 then tests whether the dequeued item corresponds to a file that is a candidate for merging. For example, some files are considered too small to be worthwhile merging, others may already be links or common store files, while others may include reparse points (described

below) that make the file ineligible for SIS merging. If the file is not a SIS candidate, step 902 branches ahead to step 916 to determine if the time is expired for this call or if there are no more items to dequeue, whereby the grovel function 96 will end. If instead the file is a SIS candidate, step 902 branches to step 904 where the file has a checksum (signature) computed therefor comprising a hash function applied to selected blocks of data (e.g., a one four kilobyte block from one-third of the way into the file, and a second four kilobyte block from two-thirds of the way into the file). As described below, this signature is computed in the same way SIS computes a signature for link files, although the present invention does not depend on this in any way, and other signature algorithms may be employed. At step 906, the file information database 100 is queried via the database manager 106 to obtain a list of files that have the same file size and signature as the file corresponding to the currently dequeued item.

If any matching files are returned as determined by step 908, step 910 takes an item from the match list and fully compares each byte in the matching file with each byte in the file corresponding to the work queue item to determine if the files are exact duplicates. Note that although not shown, some optimizations may be employed. For example, if the database manager 106 retrieves any SIS link files as matching the signature and file size, those may be tested first, actually using the common store file data for the full comparison. This is generally preferable to using the link file or a normal file because the link or normal file might be in use, in which event the groveler would have to postpone the comparison. The common store file is not busy in this way, and thus does not manifest this behavior. Moreover, to avoid a performance impact on other processes via pollution of the disk buffer cache, file reads are non-cached. To avoid interfering with foreground processes via file locking conflicts, opportunistic locks are used by the groveler 60 when accessing a file, which temporarily suspend access to the file by another process until the groveler 60 can release it.

If at step 912 a potentially matching file does not match, the process continues until the match list is empty as determined by step 908 or until an exact match is found at step 912. The first exact match that is found ends the comparison, as a merge may then take place at step 914, as described below. As shown in FIG. 9, the grovel function 96 continues to dequeue items from the work item queue 102 until the work item queue 102 is empty or the time expires.

Following the calling of the grovel function 96 at step 618, the grovel function 96 may be called as many times as needed until the partition controller 84 is halted by the central controller 82, e.g., by a call thereto or by expiration of a time setting.

As can be readily appreciated, the partition controller 84 interleaves calls to the scan volume function 94, extract log function 92 and grovel function 96 while the scan volume operation is not complete. Once the scan volume operation is complete, the extract log function 92 and grovel function 96 are interleaved to respectively add items to the work queue 102 as files are created and/or modified, and remove those items from the queue 102 in search of duplicates. The interleaving of the scan volume function 94 that adds items to the work item queue 102 with the function that removes items from the work item queue (the grovel function 96) avoids the need to allocate a very large queue 102 for the many files possible in a file system volume 62. The extract log function runs to completion once it is started in order to prevent the USN log from overflowing and thus forcing an expensive scan volume process.

SINGLE INSTANCE STORE

The present invention has been described with reference to identifying duplicate files for merging into a single instance store with links thereto, and a system for performing the merging of duplicate files is described herein. However, as can be readily appreciated, the present invention may have numerous applications in identifying files with similar properties, not only files that are exact duplicates of one another. For example, the present invention may be used to find files that although not having exactly duplicated data, are very similar to one another, e.g., have differences below some threshold number of differences. A mechanism that commonly stored the similar data as a file, with separate links thereto along with the file deltas, may be alternatively utilized to combine file data.

For efficiency, the SIS facility 66 may be built into the file system. However, although not necessary to the present invention, primarily for flexibility and to reduce complexity, it is preferable in the Windows 2000 environment to implement the SIS facility 66 as a filter driver 66' (FIG. 12). Indeed, the present invention was implemented without changing the Windows NT® file system (NTFS). Notwithstanding, it will be understood that the present invention as described above is not limited to the NTFS filter driver model.

In the NTFS environment, filter drivers are independent, loadable drivers through which file system I/O (input/output) request packets (IRPs) are passed. Each IRP corresponds to a request to perform a specific file system operation, such as read, write, open, close or delete, along with information related to that request, e.g., identifying the file data to read. A filter driver may perform actions to an IRP as it passes therethrough, including modifying the IRP's data, aborting its completion and/or changing its returned completion status.

The SIS link files 76-78 do not include the original file data, thereby reclaiming disk space. More particularly, the link files are NTFS sparse files, which are files that generally appear to be normal files but do not have the entire amount of physical disk space allocated therefor, and may be extended without reserving disk space to handle the extension. Reads to unallocated regions of sparse files return zeros, while writes cause physical space to be allocated. Regions may be deallocated using an I/O control call, subject to granularity restrictions. Another I/O control call returns a description of the allocated and unallocated regions of the file.

The link files 76, 78 include a relatively small amount of data in respective reparse points 110, 112, each reparse point being a generalization of a symbolic link added to a file via an I/O control call. As generally shown in FIG. 10, a reparse point (e.g., 110) includes a tag 114 and reparse data 116. The tag 114 is a thirty-two bit number identifying the type of reparse point, i.e., SIS. The reparse data 116 is a variable-length block of data defined by and specific to the facility that uses the reparse point 110, i.e., SIS-specific data, as described below.

FIGS. 11A-11B represent the general flow of operation when the groveler 60 makes a SIS_MERGE_FILES control request 64 to SIS to merge duplicates files via the SIS driver 66'. The SIS driver 66' receives such requests, and at step 1100 determines whether the matching file is already a SIS link file. If the matching file is a SIS link file, step 1100 branches to step 1104 to handle the merge depending on whether the file corresponding to the work item is also a SIS link, as described below.

In the event that the matching file is not a SIS link, step 1100 branches to step 1102 to determine if the file corre-

sponding to the work item is a SIS link. If not, step 1102 branches to step 1106 where the contents of the matching file are copied as file data 118 to a newly allocated file (e.g., 74) in the common store 80 (FIG. 2A). Note that for efficiency, SIS (and/or the groveler 60) may employ some threshold size test before making the copy. Further, note that SIS_MERGE_FILES control does an actual copy of the contents of the matching file to the common store 80 rather than a rename of the matching file. The link file representing the matching file thus maintains the file identifier (File ID) number originally assigned by the NTFS to the matching file, so that user open requests directed to the NTFS file ID are to the link file rather than to the common store file. This file ID number is used by SIS to identify the file, whereby any user-renaming of the link file by the user is not an issue. In an alternate embodiment, SIS could use rename in order to avoid copying the file data, possibly at the cost of having the source file's file ID change because of the copy operation, or by having support for a rename operation that leaves the file IDs unchanged in the underlying NTFS 130.

The common store file 74 in the common store 80 is named based upon a 128-bit universal unique identifier (UUID), shown in FIGS. 2A-2B as the file CommonStore\ (UUID,). Using a UUID is particularly beneficial when backing up and restoring SIS files, since files with the same UUIDs are known to be exact copies, and more than one such copy is not needed in the common store 80.

While not shown in FIG. 11A, if a copying error occurs, the matching file remains unchanged, an appropriate error message is returned to the requesting user, and the SIS_MERGE_FILES control 64 is terminated. In the normal event where there are no errors in the copying process, step 1106 continues to step 1108 where the matching file is converted to a SIS link file (e.g., the link file 76, FIG. 2B).

To convert a file to a SIS link file at step 1108, the SIS_MERGE_FILES control 64 provides the reparse point 110, including the SIS tag 114, and reparse data 116 including the common store file's unique file identifier 120 and a signature 122 (FIG. 10). The signature 122 is a 64-bit checksum computed by applying a trinomial hash function (known as the 131-hash) to the file data. The common store file 74 maintains the signature therewith as part of a backpointer stream 124, described below. The only way to determine the signature is via the file data contents, and thus the signature may be used to provide security by preventing unauthorized access to the contents via non-SIS created reparse points as described below.

As another part of the conversion to a link file 76 at step 1108, the data of the file is cleared out using the aforementioned NTFS sparse file technology. The resulting link file 76 thus essentially comprises the reparse point 110 and a shell for the data. At step 1110, the file 78 is created for the file corresponding to the work item in the same general manner, i.e., the link file 78 comprises a reparse point 112 having the same information therein and a shell for the data. Each link file is on the order of approximately 300 bytes in size.

Step 1114 represents the adding of identifiers of any new link files (converted via steps 1108 and/or 1110) to a backpointer stream 124 maintained in the common store file 74. As described in more detail below, the backpointers identify to the common store file 74 the link files that point to it. As also described below, backpointers are particularly useful in delete operations, i.e., delete the backpointer when the link file is deleted, but only delete the common store file when it has no more backpointers listed in the stream 124. At this time, the common store file 74 and the links 76, 78

13

thereto are ready for use as SIS files, and the files are closed as appropriate (step 1116).

Alternatively, if at step 1102 the file corresponding to the work item is already a SIS link file, there is no need to create another common store file to merge the files. Step 1102 thus branches to step 1112 where the matching file is converted to a link file as described above with reference to step 1108. Step 1112 then continues to step 1114 to add the backpointer of the link file converted from the matching file to the common store file, and then the files are closed as appropriate at step 1116.

Returning to step 1100, in the event that the matching file is a SIS link file, step 1100 branches ahead to step 1104 to determine if the file corresponding to the work item is also a SIS link file. If at step 1104 the file corresponding to the work item is not a SIS link, step 1104 branches to perform steps 1110-1116 to convert the file corresponding to the work item to a SIS link file in the manner described above. If instead the file already is a SIS link file, i.e., both files are SIS link files, step 1104 branches to step 1120 of FIG. 11B.

At step 1120 of FIG. 11B, the link files are evaluated to determine if they refer to the same common store file. If so, there is nothing to merge, and thus the process ends by returning to step 1116 (FIG. 11A) and closing any files as appropriate. If the two files do not refer to the same common store file, then the work item's corresponding file is converted to point to the common store file to which the matching store file points, and the corresponding common store files are appropriately modified. Note that this situation is possible if the user creates links by using the SIS_COPYFILE method.

More particularly, to properly handle the conversion of the work item's corresponding file and the fixup of the common store files, as represented by step 1122, the backpointer for the file corresponding to the work item is removed from its corresponding common store file. To adjust the file corresponding to the work item, the reparse point of this link file is converted to point to the common store file referred to by the matching file, as represented by step 1124. Also, as represented by step 1126, a backpointer to the file that corresponds to the work item is added to the common store file that is referred to by the matching link file. The process then returns to FIG. 11A to close the files as appropriate at step 1116.

Turning to FIGS. 12 and 13, there is provided an explanation of how a request to open a link file is handled by the SIS/NTFS architecture. As shown in FIG. 12, an open request in the form of an IRP, (including a file name of a file that has a SIS reparse point), as represented by the arrow with circled numeral one, comes in as a file I/O operation and is passed through a driver stack. The driver stack includes the SIS filter driver 66' with other optional filter drivers 126, 128 possibly above and/or below the SIS filter driver 66'. For purposes of the examples herein, these other filter drivers 126, 128 (shown herein for completeness) do not modify the IRPs with respect to SIS-related IRPs. At this time, the SIS filter driver 66' passes the IRP on without taking any action with respect thereto, as it is generally not possible to determine if a given filename corresponds to a file with a reparse point until NTFS processes the open request.

When the SIS link open IRP reaches the NTFS 130, the NTFS 130 recognizes that the file named in the IRP has a reparse point associated therewith. Without further instruction, the NTFS 130 does not open files with reparse points. Instead, the NTFS 130 returns the IRP with a STATUS_REPARSE completion error and with the con-

14

tents of the reparse point attached, by sending the IRP back up the driver stack, as represented in FIG. 12 by the arrow with circled numeral two. As represented in FIG. 13A, at step 1300 the SIS filter 66' receives the STATUS_REPARSE error and recognizes the IRP as having a SIS reparse point.

In response, via steps 1302-1304, the SIS filter 66' opens the common store file 74 identified in the reparse point if the common store file 74 is not already open, and reads the signature therein. This is accomplished by the SIS filter 66' sending separate IRPs to NTFS 130 identifying the common store file by its UUID name 120 (FIG. 10) in the reparse point 110, and then requesting a read of the appropriate data. Then, at step 1306, if the open proceeded correctly, the SIS filter 66' compares the signature 122 in the reparse point with the signature in the backpointer stream 124 of the common store file 74. If they match, step 1306 branches to step 1320 of FIG. 13B as described below. However, if the signatures do not match, the SIS filter 66' allows the open to proceed by returning a file handle to the link file to the user, but without attaching SIS context to the opened file, essentially denying access to the common store file 74 for security reasons.

More particularly, a SIS reparse point may be generated external to SIS, including the UUID-based name of a common store file, a name which can be guessed in a relatively straightforward manner. As a result, without the signature check, such an externally-generated reparse point could give potentially unauthorized access to the common store file. However, since the SIS-reparse point has a signature, and the signature may only be computed by having access to the file data, only those who already have access to the file data can know the signature and provide a valid SIS-reparse point. The file data in the common store is thus as secure as the file data was in the original source file.

If the signature does not match at step 1306, step 1308 returns access to the link file without corresponding access to the common store file to the user. Step 1310 then tests to see if another link file has the common store file open, and if not, step 1312 closes the common store file 74. More particularly, SIS maintains a data object that represents the common store file, and the common store file data object keeps a reference count of open link files having a reference thereto. Step 1310 essentially decrements the reference count and checks to see if it is zero to determine whether it needs to close the common store file handle. Note that valid users are thus not stopped from working with their valid links to the common store file 74 if an invalid reparse point is encountered during the valid users' sessions.

If the signatures match at step 1306, at step 1320 the SIS filter driver 66' sets a FILE_OPEN_REPARSE_POINT flag in the original link file open IRP, and returns the IRP to the NTFS 130, as shown in FIG. 12 by the arrow with circled numeral three. This flag essentially instructs the NTFS 130 to open the link file 76 despite the reparse point. As shown in FIG. 12 by the arrow with circled numeral four, the NTFS 130 returns success to the SIS filter 66' along with a file object having a handle thereto (assuming the open was successful). At step 1322 of FIG. 13B, when the success is received, the SIS filter driver 66' attaches context 132 (FIG. 2B) to the file object, including a context map 134 (FIG. 10) that will be used to indicate any portions of the link file that have been allocated to data. Note that the context 132 is an in memory structure and only attached while the file is open, and is thus represented by a dashed box in FIG. 2B to reflect its transient nature. If the link file has any allocated data portions, those portions are marked in the map 134 in the

15

context as "dirty" at step 1322. A link file having allocated data when first opened is a special case situation that occurs, for example, when the disk volume 62 was full, as described below.

At step 1326, a check is made to ensure that the link file's identifier is listed among the backpointers in the backpointer stream 124 of the common store file 74. It is possible for the list of backpointers in the stream 124 to become corrupted (e.g., when the SIS filter driver 66' is not installed) whereby the link file 76 is not listed. If not listed at step 1326, the link file's identifier, which is known to identify a valid link, is added to the list of backpointers 124 at step 1328, and a volume check procedure 136 (FIG. 2B) is started at step 1330 (unless already running). The volume check 136 essentially works with the backpointer streams of the various common store files (UUID₁-UUID_n) so that common store files do not contain backpointers to link files that do not exist, so that common store files do not remain and use disk space without at least one link pointing thereto, and so that each valid link file has a backpointer in the corresponding common store file. At step 1332, if volume check 136 is running, a check bit, used by the volume check 136, is set to one in the backpointer for the file each time that link file is opened. The volume check 136 and check bit are described in the aforementioned copending United States Patent Application entitled "Single Instance Store for File Systems."

At step 1334, the handle to the link file is returned to the user, shown in FIG. 12 by the arrow with circled numeral five. Note that the user thus works with the link file 76, and generally has no idea that the link file 76 links the file to the common store file 74. At this time, assuming the signature was correct and the opens were successful, the user has a handle to the link file 76 and the common store file 74 is open.

Writing to a SIS link file 76 does not change the common store file 74, since other links to the common store file 74 are logically separate. Instead, write requests are written to space allocated therefor in the link file 76, as described below. In this manner, changing the data via one link does not result in changes seen by the other links. Thus, by "logically separate" it is meant that in a SIS link, changes made to one link file are not seen by users of another link file, in contrast to simply having separate file names, protections, attributes and so on. If two users open the same link file, they will see one another's changes.

FIGS. 14 and 15 describe how the SIS filter 66' handles a write request to the open link file 76. As shown in FIG. 14, the SIS write request comes through the driver stack to the SIS filter driver 66' as an IRP, including the file handle and attached context 132. The IRP designates the region of the file to be written and identifies the location of the data to write. The SIS filter driver 66' can recognize the context 132 as belonging to SIS, but because the write is directed to the link file 76, SIS lets the IRP pass to the NTFS 130 as shown in FIG. 14 by the arrow with circled numeral one and in FIG. 15 as step 1500. NTFS attempts the write, allocating appropriate space in the link file 76, and SIS receives a status from the NTFS at step 1502 (the arrow with circled numeral two in FIG. 14). If the write failed, e.g., the disk is full and the space could not be allocated, step 1504 branches to step 1506 where the error is returned to inform the user.

If the write was successful, step 1504 branches to step 1508 where the SIS filter driver 66' marks the region that was written as dirty in the context map 134 of the context 132, while step 1510 then reports the successful write status to the user. In this manner, SIS tracks which part of the file

16

data is current in the common store file 74 and which part is current in the link file 76. By way of example, consider a user requesting to write ten kilobytes of data beginning at offset one megabyte, as generally shown in FIG. 10. The NTFS 130 allocates the space, unless already allocated, in the appropriate region 138 of the link file's (sparse) data space 140 (note that the NTFS actually allocates space in 64-kilobyte blocks). SIS then marks the context map 134 to reflect this dirty region, as shown in FIG. 10. Note that since the changes are not written to the common store file 74, the changes written to one link file are not seen by any other link to the common store file 74.

SIS thus lets NTFS 130 handle the allocation of the space in the sparse file and the writing thereto. However, if SIS is implemented in a file system that did not have sparse file capabilities, SIS could perform the equivalent operation by intercepting the write request and writing the data to a temporary file. Upon closing the "changed" link file, SIS only need copy the clean data from the common store file to the temporary file, delete the link file and rename the temporary file with the name of the link file to achieve the logical separation of files in a transparent manner.

FIGS. 16 and 17 describe how the SIS filter 66' handles a read request to the open link file 76. As shown in FIG. 16, the SIS read request comes through the driver stack to the SIS filter driver 66' as an IRP, including the file handle and attached context. The SIS filter driver 66' recognizes the attached context 132 as belonging to SIS, and intercepts the IRP, shown in FIG. 16 by the arrow with circled numeral one.

As shown in step 1700 of FIG. 17, the SIS filter driver initially examines the map 134 in the attached context 132 to determine if any of the link file is marked as dirty, i.e., allocated to file data. Step 1702 then compares the region that the IRP is specifying to read against the map 134, and if the read is to a clean region, step 1702 branches to step 1704. At step 1704, SIS converts the link file read request to a common store file read request IRP and passes the modified IRP to the NTFS 130 as also shown by the arrow accompanied by the circled numeral 2a in FIG. 16. The NTFS 130 responds with the requested data (or an error) as shown in FIG. 16 by the arrow with circled numeral 3a. The data (or error) is then returned to the user at step 1716 of FIG. 17, (circled numeral 4 in FIG. 16). Note that to the user, the request appears to have been satisfied via a read to the link file, when in actuality the SIS filter 66' intercepted the request and converted it to a request to read from the common store file 74.

Returning to step 1702, it is possible that via a write operation to the link file, some of the data requested to be read is from a "dirty" region, that is, one that has been allocated and written to while the link file was open (or that was allocated on the disk when the link was first opened in step 1322). As described above, write requests cause space to be allocated in the link file 76 to provide an actual region to maintain the current state of the changed data. At step 1702, if a requested region to read is marked as dirty, step 1702 branches to step 1706 to determine if the entire read is from a dirty region or spans both dirty and clean regions.

If the entire region is dirty, then the SIS filter 66' passes the read request IRP to the NTFS 130 whereby the link file 76 is read at step 1708 and returned to the SIS filter 66'. This is represented in FIG. 16 by the arrows designated with circled numerals 2b and 3b. The data (or error) is then returned to the user at step 1716 of FIG. 17, (circled numeral 4 in FIG. 16). In this manner, the user receives the current changes that have been written to the link file rather than the stale data in the common store file 74.

17

Alternatively, if step 1706 detects that the user is requesting both clean and dirty regions, the SIS filter 66' splits up the read request into appropriate requests to read the dirty region or regions from the link file 76 and the clean region or regions from the common store file 74. To this end, at steps 1710 and 1712, the SIS filter 66' uses the map 134 to generate one or more IRPs directed to reading the common store file 74 and passes at least one IRP directed to reading the link file 76 and at least one IRP directed to reading the common store file 74 to the NTFS 130. This is represented in FIG. 16 by arrows labeled with circled numerals 2a and 2b. Assuming no read errors, step 1714 merges the read results returned from the NTFS 130 (in FIG. 16, the arrows labeled with circled numerals 3a and 3b) into a single result returned to the user at step 1716 (the arrow labeled with circled numeral 4). Note that any read error will result in an error returned to the user, although of course SIS may first retry on an error. By appropriately returning the current data in response to a read request from either the common store file 74 or the link file 76, or both, SIS maintains the logical separation of the link files in a manner that is transparent to the requesting user.

FIG. 18 represents the steps taken when a request to close the handle to the link file 76 is received and the handle is closed at step 1800. At step 1802, a test is performed to see if this was the last handle currently open to this link file. If not, the process ends, whereby the link file is left open for operations via the other open file handles. If instead this was the last open handle, step 1804 makes a determination (via the context map 134) if any portion of the link file 76 is marked as dirty (allocated). If not, the driver 66' requests closing of the common store file handle, whereby steps 1806 and 1808 cause the common store file 74 to be closed if no other links have the common store file 74 open, otherwise the common store file 74 remains open for the other links to use. Conversely, at step 1804, if any region of the link file 76 was written to and is thus marked as dirty, step 1804 branches to step 1810 since the link file may no longer be properly represented by the common store file 74. Note that steps 1810 and below may take place after the link file handle has been closed, by doing the work in a special system context. This allows the users to access the SIS file while the copyout of clean data is in progress. Step 1810 copies the clean portions from the common store file 74 to space allocated therefor in the link file 76. If successful at step 1812, the now fully-allocated link file is converted back to a regular file at step 1814, essentially by removing the reparse point. In this manner, logically independent links to the common store file are supported, as the changes made to one link file are not seen via any other link file. The link file 76 is then deleted from the list of files in the backpointer stream as described below with reference to FIG. 19, which may further result in the common store file being deleted. The process then continues to steps 1806 and 1808 to close the common store file if no other links have it open. Note that the handle to the common store file needs to be closed even if the common store file was deleted.

However, it is possible that the clean data from the common store file 74 could not be copied back, particularly if the space therefor could not be allocated in the link file 76 due to a disk full condition. If such an error occurs, step 1812 branches to step 1816 which represents the canceling of the copyout and leaving the link file 76 as is, preserving the written data. Note that this will not cause a disk full condition because the space was already allocated to the link file during the earlier write request without an error, otherwise the write request that caused the space to be allocated

18

would have failed and the user notified (FIG. 15, steps 1504-1506). As described above, when the link file is re-opened, step 1322 of FIG. 13B will mark the allocated portions of the link file 76 as dirty in the map 134, whereby the changes are properly returned when the file is read. Step 1816 then continues to steps 1806 and 1808 to close the common store file if no other links have it open.

In a similar manner to the disk full condition, it is thus possible in general to employ the SIS architecture to use the link file 76 to maintain changes (deltas), with the unchanged clean regions backed up by the common store file 74. To this end, instead of copying the clean portions from the common store file and reconvert the link file to a regular file when the file is closed, SIS may keep the link file as a link file with whatever space is allocated thereto. Some criteria also may be used to determine when it is better to convert the link file back to a regular file. For example, a threshold test as to the space saved may be employed to determine when to return a link file to a regular file versus keeping it as a link, whereby only link files with relatively small deltas would be maintained as link files. As a result, SIS may provide space savings with files that are not exact duplicates, particularly if the file contents are almost exactly identical. As mentioned above, the groveler 60 may also identify near-duplicate files for merging in this manner. Notwithstanding, at present SIS preferably employs the copy-on-close technique of FIG. 18, since updates of SIS files and/or writes thereto are likely to be relatively rare.

Turning to FIG. 19, there is shown a process employed by SIS after a link file is deleted (e.g., by file I/O) or reconverted to a regular file (e.g., by the SIS close process). When a SIS link is deleted or reconverted to a regular file, the common store file 74 corresponding to that SIS link file is not necessarily deleted because other links may be pointing to that common store file 74. Thus, at step 1902, the backpointer stream 124 is evaluated to determine if the deleted backpointer was the last backpointer remaining in the stream, i.e., there are no more backpointers. If it is not the last backpointer, then there is at least one other link file pointing to the common store file 74, the common store file 74 is thus still needed, and the process ends. In this manner, logically independent links to the common store file are again supported, as deleting one link file does not affect any other link file.

If no backpointers remain at step 1902, this generally indicates that no link files are pointing to the common store file and thus the common store file is no longer needed. However, before deleting the common store file, step 1902 branches to step 1904 where a test is performed as to whether the volume check procedure 136 is running. If so, there is a possibility that the backpointer stream is corrupted, as described below. If the volume check is not currently running, step 1904 advances to step 1908 to delete the common store file (after first closing it, if necessary). Otherwise, since the backpointer stream is not necessarily trustworthy, step 1904 branches to step 1906 where it is determined whether the volume check 136 is calling this delete procedure, i.e., whether the steps of FIG. 19 are being invoked from the volume check. If the volume check is not calling to delete the file, step 1906 ends the process without deleting the file, otherwise step 1906 branches to step 1908 to delete the file. Step 1906 thus enables the volume check 136 to delete a common store file when the volume check has concluded that the backpointer stream is correct and no link files point thereto.

In sum, step 1908 deletes the common store file when the backpointer stream is both empty and trusted, thereby

reclaiming the disk space. Note that instead of backpointers, counts of the links may be alternatively used for this purpose, i.e., delete the common store file when a count of zero links thereto remain. Backpointers are preferable, however, primarily because they are more robust than counts.

As can be seen from the foregoing detailed description, there is provided a method and system that provide for the identifying and merging of duplicate files. The method and system may operate dynamically as a real time background process, in an efficient manner.

While the invention is susceptible to various modifications and alternative constructions, a certain illustrated embodiment thereof is shown in the drawings and has been described above in detail. It should be understood, however, that there is no intention to limit the invention to the specific form or forms disclosed, but on the contrary, the intention is to cover all modifications, alternative constructions, and equivalents falling within the spirit and scope of the invention.

What is claimed is:

1. A computer-readable medium having computer-executable instructions, comprising, automatically identifying at least two files having duplicate data, automatically merging the duplicate data of the files into a single instance representation of that data, converting each of the files into logically separate links to the single instance representation, each link comprising a logically separate link file that provides logically separate file system access to the single instance representation of the file data, and reclaiming storage space that was occupied by the duplicate data of at least one of the files.
2. The computer-readable medium having computer-executable instructions of claim 1 wherein automatically identifying at least two files having duplicate data includes, adding file identifiers to a work item queue.
3. The computer-readable medium having computer-executable instructions of claim 2 wherein adding file identifiers to a work item queue includes scanning a volume for file identifiers.
4. The computer-readable medium having computer-executable instructions of claim 3 wherein scanning the volume for file identifiers occurs for a limited time.
5. The computer-readable medium having computer-executable instructions of claim 2 wherein adding file identifiers to a work item queue includes extracting file information from a log of file activity.
6. The computer-readable medium of claim 5 having further computer-executable instructions for calculating a time for extracting file information from the log.
7. The computer-readable medium having computer-executable instructions of claim 6 wherein the time calculated is based on an amount of file information previously extracted from the log.
8. The computer-readable medium having computer-executable instructions of claim 1 wherein automatically identifying at least two files having duplicate data includes, dequeuing a file identifier from a work item queue.
9. The computer-readable medium of claim 8 having further computer-executable instructions for, querying a database of file information for a set of at least one file having properties that match properties of a file corresponding to the identifier dequeued from the work item queue.
10. The computer-readable medium having computer-executable instructions of claim 9 wherein querying the database of file information includes providing a file size of the file corresponding to the identifier dequeued from the work item queue to a database manager.

11. The computer-readable medium of claim 9 having further computer-executable instructions for, calculating a signature of the file corresponding to the identifier dequeued from the work item queue.

12. The computer-readable medium having computer-executable instructions of claim 11 wherein querying the database of file information includes providing the signature to a database manager.

13. The computer-readable medium having computer-executable instructions of claim 12 wherein querying the database of file information further includes providing a file size of the file corresponding to the identifier dequeued from the work item queue to the database manager.

14. The computer-readable medium of claim 9 having further computer-executable instructions for, receiving the set of at least one file having properties that match properties of the file corresponding to the identifier dequeued from the work item queue, and comparing the data of at least one file in the set with the data of the file corresponding to the identifier dequeued from the work item queue.

15. The computer-readable medium having computer-executable instructions of claim 14 wherein comparing the data determines if each file is an exact duplicate of the other.

16. A method of identifying files having similar properties on a file system volume, comprising, in a first operation, adding file information to a queue, in a second operation, removing file information from the queue, querying a database with at least one property of a file corresponding to the file information removed from the queue, and receiving a set of at least one file identifier, each file identifier in the set corresponding to a file having at least one similar property of the file corresponding to the file information removed from the queue.

17. The method of claim 16 wherein adding file information to a work item queue includes scanning a volume for file identifier information.

18. The method of claim 17 further comprising limiting the time for scanning the volume.

19. The method of claim 17 wherein adding file information to a work item queue includes extracting file information from a log of file activity.

20. The method of claim 19 further comprising calculating a time for extracting file information from the log.

21. The method of claim 20 further comprising, returning an amount of file information extracted from the log, and using the amount to calculate a next time for extracting file information from the log.

22. The method of claim 16 further comprising calculating a signature of the file corresponding to the file information removed from the queue, and wherein querying the database includes providing the signature to a database manager.

23. The method of claim 22 wherein querying the database further includes providing a file size corresponding to the file information removed from the queue to the database manager.

24. The method of claim 16 wherein querying the database includes providing a file size corresponding to the file information removed from the queue.

25. The method of claim 16 further comprising, comparing data in the file that corresponds to the file information removed from the queue to the data in at least one file corresponding to file identifier information in the set, and if sufficiently similar, merging the files into a single instance representation thereof having independent links thereto.

26. The method of claim 25 wherein comparing the data determines if each file is an exact duplicate of the other.

21

27. A system for identifying files having similar properties on a file system volume, comprising, a database including file property information, a database manager for querying the database, a work queue, a first component for adding file identifiers to the work queue, and a second component for removing file identifiers from the queue, the second component providing a query to the database manager, the query including property information corresponding to a file identified by a file identifier removed from the queue, the second component receiving a set of file identifiers in response to the query, each identifier in the set corresponding to a file having property information that matches the file property information identified in the query.

28. The system of claim 27 wherein the second component compares the data of the file corresponding to the file identifier removed from the queue with the data of at least one file corresponding to a file identifier returned in response to the query.

29. The system of claim 28 wherein the second component performs a byte comparison of the data in each file to determine if the file data matches exactly.

30. The system of claim 27 wherein if the comparison indicates the file data is similar, the second component calls a facility for merging the files, the facility providing a single instance representation of the file data and logically separate links thereto.

31. The system of claim 27 further comprising a log for recording file activity, and wherein the first component extracts at least some of the file identifiers for adding to the work queue from the log.

22

32. The system of claim 27 further comprising a third component for scanning a volume to add file identifiers to the queue.

33. The system of claim 27 wherein the file property information includes a file size.

34. The system of claim 27 wherein the file property information includes a signature.

35. The system of claim 34 wherein the second component computes a signature of the file corresponding to the file removed from the queue.

36. The system of claim 27 wherein the first and second components are functions within a single process, and wherein a partition controller corresponding to a file system volume calls the functions.

37. The system of claim 36 including a plurality of partition controllers, each partition controller corresponding to a file system volume, and further comprising a central controller for controlling the operation of the partition controllers.

38. The system of claim 37 wherein the central controller operates the partition controllers as a background process.

39. The method of claim 16 wherein the first operation alternates with the second operation.

40. The method of claim 39 wherein the first operation operates for a limited time, and the second operation operates after the time to remove each set of file information added to the queue in the first operation.

41. A computer-readable medium having computer-executable instruction for performing the method of claim 16.

* * * * *



US006640317B1

(12) **United States Patent**
Snow

(10) **Patent No.:** **US 6,640,317 B1**
(45) **Date of Patent:** **Oct. 28, 2003**

(54) **MECHANISM FOR AUTOMATED GENERIC APPLICATION DAMAGE DETECTION AND REPAIR IN STRONGLY ENCAPSULATED APPLICATION**

(75) **Inventor:** **Paul Alan Snow, Richland, WA (US)**

(73) **Assignee:** **International Business Machines Corporation, Armonk, NY (US)**

(*) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) **Appl. No.:** **09/552,862**

(22) **Filed:** **Apr. 20, 2000**

(51) **Int. Cl.:** **H02H 3/05; G06F 9/44**

(52) **U.S. Cl.:** **714/38; 717/124; 717/174**

(58) **Field of Search:** **714/6, 15, 38, 714/39, 42, 44; 707/103 R; 717/168-178, 124**

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,711,863 A	*	1/1973	Bloom	714/38
5,001,628 A	*	3/1991	Johnson et al.	364/200
5,497,484 A	*	3/1996	Potter et al.	395/600
5,745,669 A	*	4/1998	Hugard et al.	714/3
5,832,266 A	*	11/1998	Crow et al.	395/700
5,867,714 A	*	2/1999	Todd et al.	717/172

5,951,698 A	*	9/1999	Chen et al.	714/38
6,122,639 A	*	9/2000	Babu et al.	707/103 R
6,230,284 B1	*	5/2001	Lillevold	714/13
6,351,752 B1	*	2/2002	Cousins et al.	707/103 R
6,356,933 B2	*	3/2002	Mitchell et al.	709/203
6,480,944 B2	*	11/2002	Bradshaw et al.	711/162

* cited by examiner

Primary Examiner—Robert Beausoliel

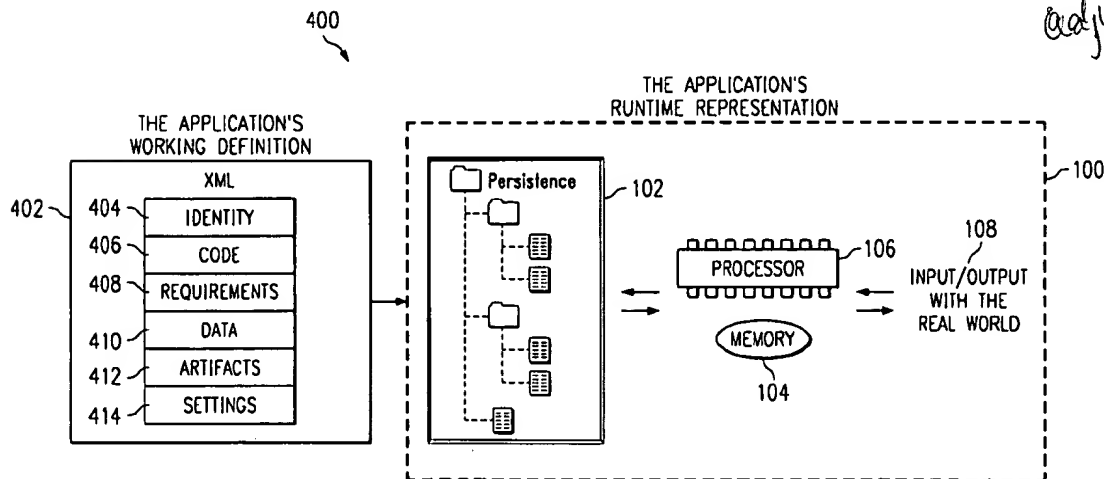
Assistant Examiner—Emerson Puente

(74) **Attorney, Agent, or Firm**—Duke W. Yee; David A. Mims, Jr.; Stephen R. Loe

(57) **ABSTRACT**

A method, system, and apparatus for detecting and repairing damaged portions of a computer system is provided. In a preferred embodiment of the present invention, a damage detection and repair facility monitors and detects changes to the computer system. The damage detection and repair facility compares these changes to the set of constraints defined by the working definitions for each application installed on the computer system. The working definitions define the invariant portions of each application and define the constraints placed upon the computer system by each application. Responsive to changes that are in conflict with this set of constraints, the damage detection and repair facility makes such changes in the persistent storage so as to resolve these conflicts. This may be done, for example, by repairing a damaged file, installing a missing driver, or adjusting an environment variable.

33 Claims, 6 Drawing Sheets



102 for claims 1-7
↓
claim 2, 5

working definition

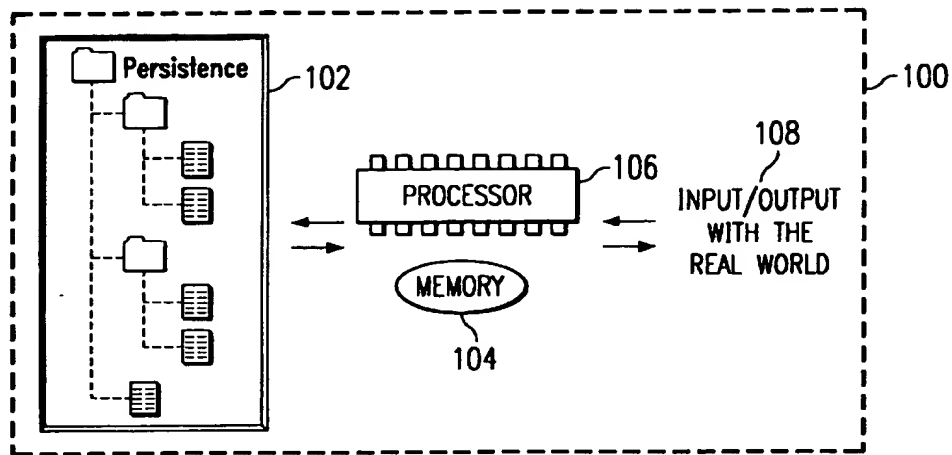


FIG. 1
(PRIOR ART)

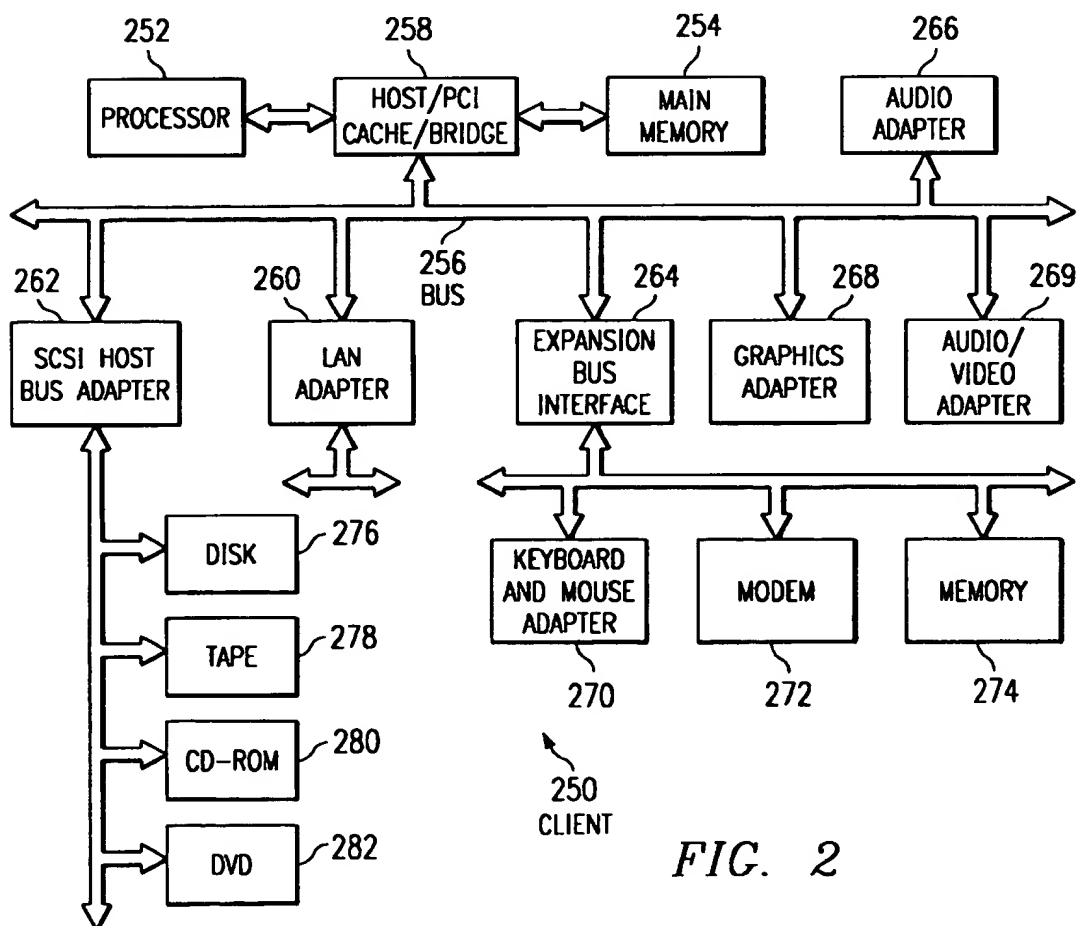


FIG. 2

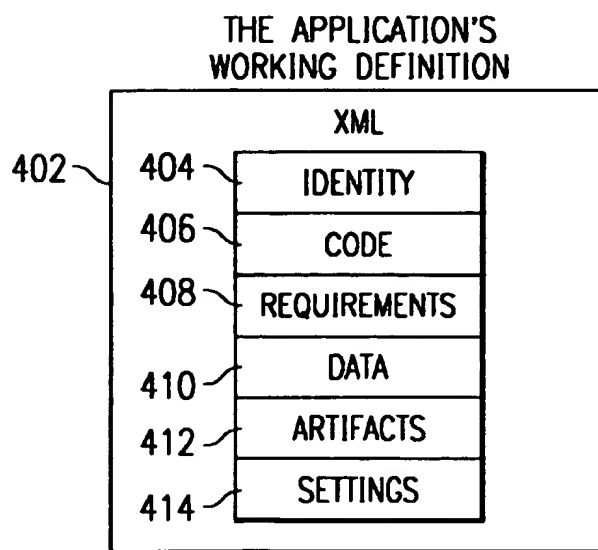
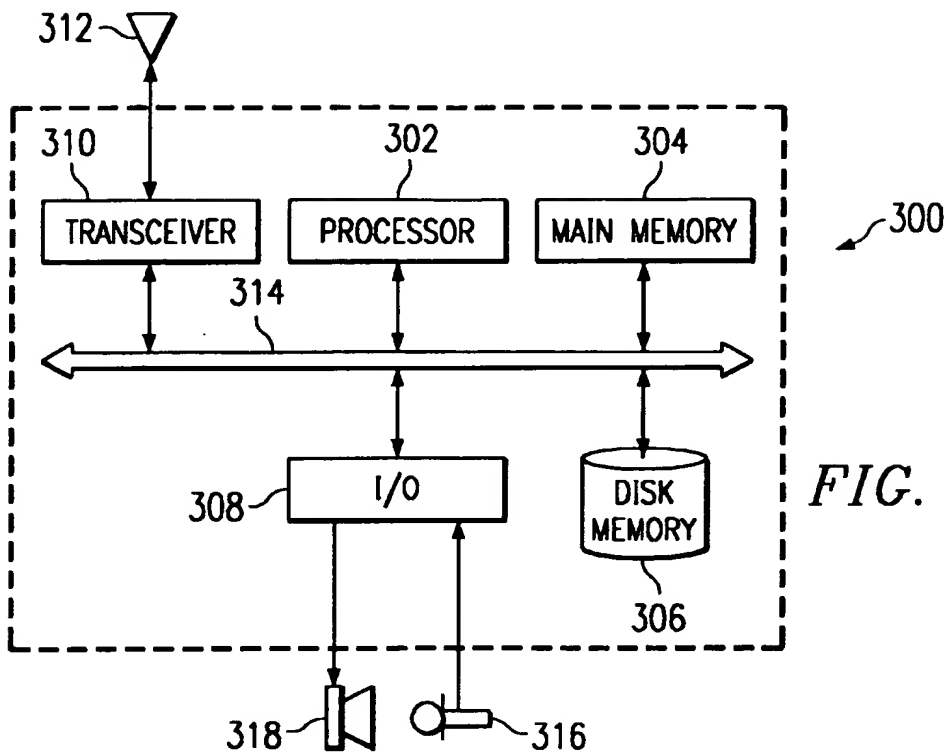
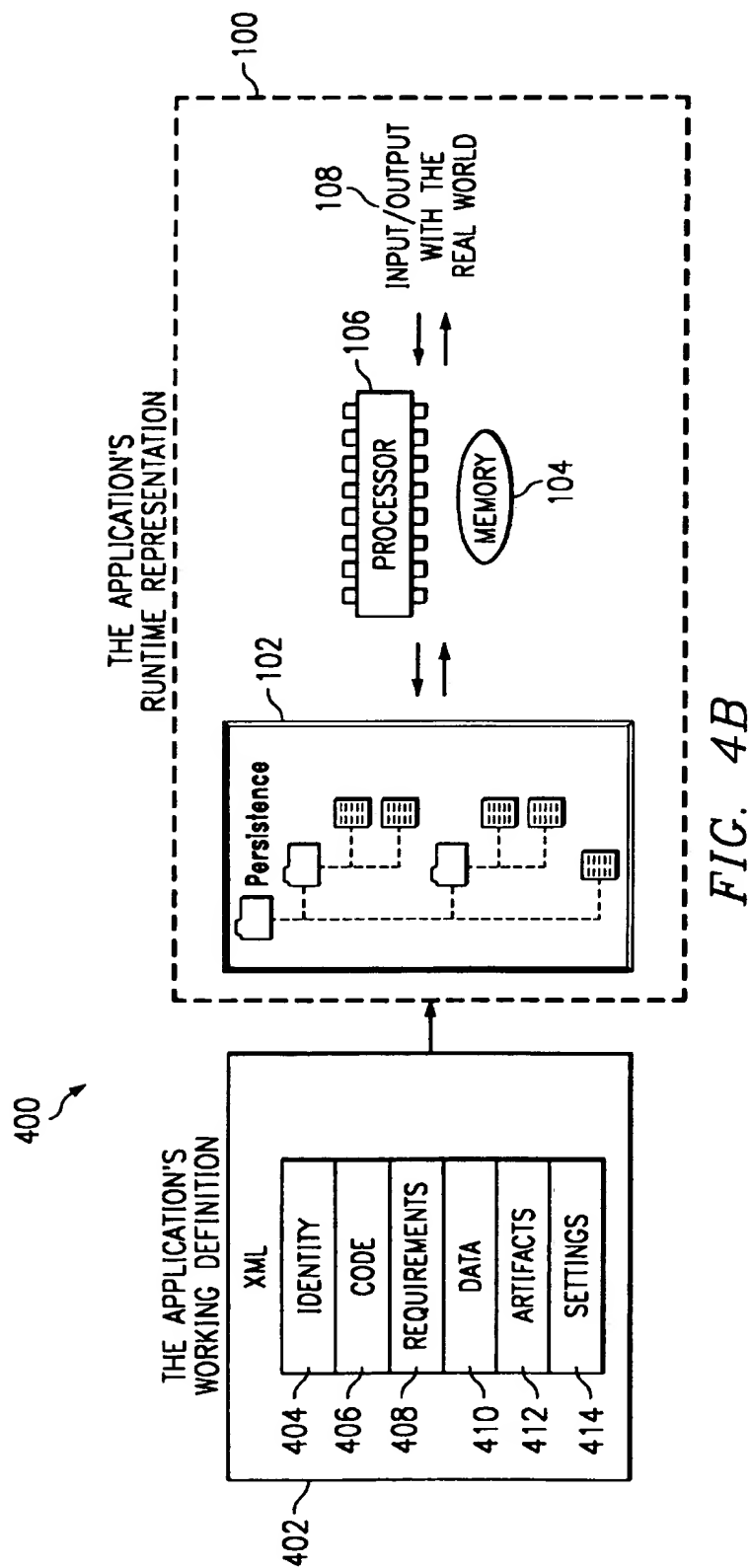


FIG. 4A



500
↙

```
<requirements>
502 { <platform>
      <processor>xx86</processor>
      <os>"Windows 95" or "Windows 98" or "Windows NT" </os>
    </platform>
504 { <services>
      <service>TCP/IP</service>
      <service>File System</service>
    </services>
506 { <prereqs>
      <application>Acrobat</application>
    </prereqs>
508 { <registryEntry name="HKEY_CURRENT_USER/Software/X.Com">
      <key>DebugPort<value>TCP/IP Port#</value></key>
      <key>Java VM<value>x<test>x gt "1.1.6"</test></value></key>
    </registryEntry>
510 { <directory name="programRoot"/>
      <directory name="jclass">programRoot"/jclass"</directory>
      <environmentVar name=path>
512 { <add>bin</add>
      </environmentVar>
      <environmentVar name=classpath>
      <add>jclass"/jappNetUI.jar"</add>
      <test>programRoot"/testClasspath"</test>
      </environmentVar>
    </requirements/>
```

FIG. 5

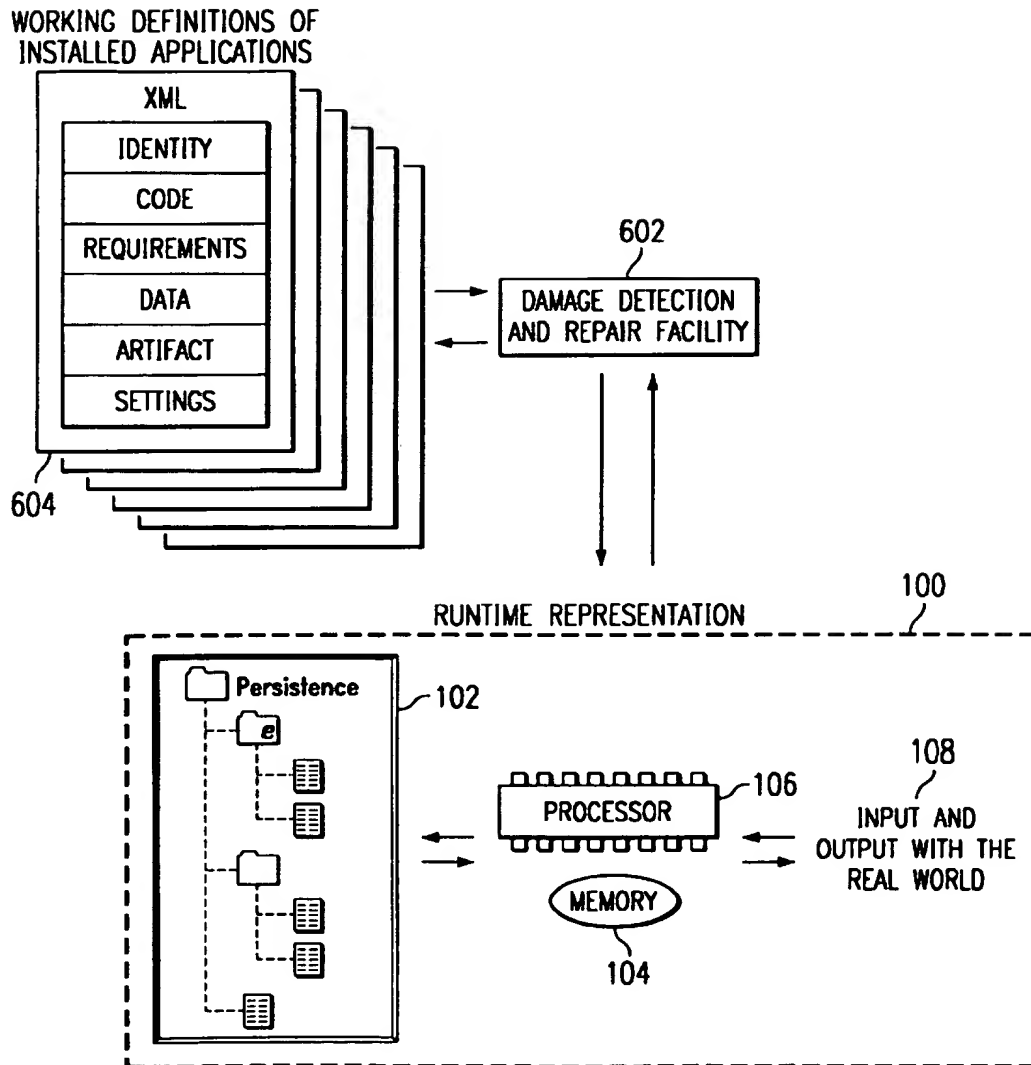


FIG. 6

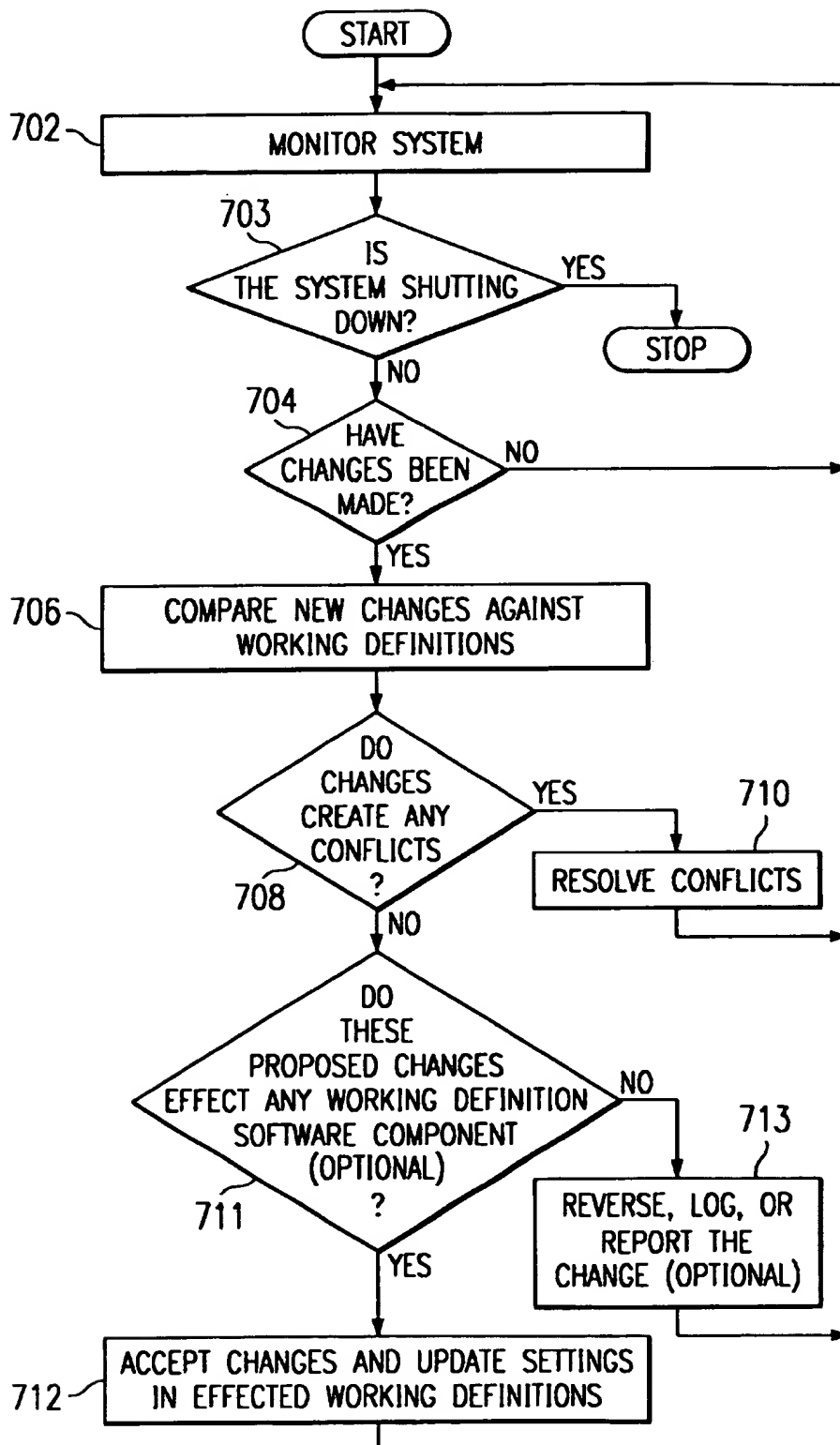


FIG. 7

MECHANISM FOR AUTOMATED GENERIC APPLICATION DAMAGE DETECTION AND REPAIR IN STRONGLY ENCAPSULATED APPLICATION

CROSS REFERENCE TO RELATED APPLICATIONS

The present application is related to co-pending U.S. patent application Ser. No. 09/552,863 entitled "Strongly Encapsulated Environments for the Development, Deployment, and Management of Complex Application configurations" filed even date herewith, now abandoned to co-pending U.S. patent application Ser. No. 09/552,864 entitled "A Method for Creating Encapsulated Applications with Controlled Separation from an Application's Runtime Representation" filed even date herewith, and to co-pending U.S. patent application Ser. No. 09/552,861 entitled "An Application Development Server and a Mechanism for Providing Different Views into the Same Constructs within a Strongly Encapsulated Environment" filed even date herewith. The content of the above mentioned commonly assigned, co-pending U.S. Patent applications are hereby incorporated herein by reference for all purposes.

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates generally to the field of computer software and, more particularly, to methods of detecting and repairing damaged files and settings within a data processing system.

2. Description of Related Art

Currently, computer systems are built up through a series of installations provided by different software developers, each of which installs one or more different software components. There is no industry standard way to describe what comprises an application. Without this description, there is no way to implement a standard service that can protect the integrity of each application.

There is a need for a standard, simple, scalable, platform independent mechanism for detecting damaged applications and repairing them. Applications can be damaged in a variety of ways. Each of the following operations can damage one or more applications within a computer system:

- Installation of new applications
- Reconfiguring applications
- The use of an application
- Application error
- User error
- Viruses

Installation is dangerous because the current approach in the industry is a rather ad hoc approach in which the responsibility for the installation procedure for each application rests with each application's developer. During installation, any error or conflict between any two or more applications may potentially corrupt the computer system configuration. Applications do not have access to the requirements and dependencies of the other applications already installed on the target computer system. Without this information, no modification to a computer system's files and settings can be made completely safe. Yet the modification of files and settings is required to install applications onto computer systems.

The reconfiguring of an application may also require that changes be made to files and configuration settings needed

or used by other applications, and such changes may render one or more of the other applications inoperable.

The use of applications may also require that the files and settings within a computer system be updated from time to time. This requires applications to be "well behaved" with respect to each other. Conflicts may still occur, either by chance, or error within an application. Yet the application developer's priority is always to their application without regard to potential conflicts with other applications, except to insure their application wins such conflicts. Because the requirements and constraints of each application are not defined, each developer also becomes responsible for supporting the configuration management of every system on which the application is installed, and for handling conflicts in all configurations in which their application may be used. However, this responsibility is rarely, if ever, each application developer's top priority.

The current system of deploying and running applications defines only very weak barriers between applications to prevent a given application from damaging other applications. Any conflict, if handled in the given application's favor, may prevent a competitor's application from running. This state of affairs provides little motivation to avoid conflicts with a competitor's application so long as the developer's own application can be configured properly. Each application is dependent on its own installation and runtime code to verify its configuration and access to prerequisite devices, drivers, applications, and system settings.

Users themselves may cause problems by modifying files or configuration settings, either by accident, failure to follow a procedure properly, etc. Often all the files and configuration settings are exposed to modification by the user.

In reality, virus attacks account for a very small percentage of all application failures. Yet a viral attack can result in a huge amount of damage.

Currently the generally accepted method of protecting a computer system from damage from outside viral attacks is through the use of third party vendor virus protection software products. However, these products must be updated frequently to be able to detect the latest viruses. Any new virus that has been created since the software was updated may be undetected by the virus protection software, thus enabling that virus to corrupt or destroy files necessary for the proper performance of the computer.

Therefore, there is a need for a method, system and apparatus that automatically detects damaged files and applications and restore them to their proper condition.

SUMMARY OF THE INVENTION

The present application provides a method, system, and apparatus for detecting and repairing damaged portions of a computer system. In a preferred embodiment of the present invention, a damage detection and repair facility monitors and detects changes to the computer system. The damage detection and repair facility compares these changes to the set of constraints defined by the working definitions for each application installed on the computer system. The working definitions define the invariant portions of each application and define the constraints placed upon the computer system by each application. Responsive to changes that are in conflict with this set of constraints, the damage detection and repair facility makes such changes in the persistent storage so as to resolve these conflicts. This may be done, for example, by repairing a damaged file, installing a missing driver, or adjusting an environment variable.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself,

however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

FIG. 1 depicts a block diagram illustrating a basic prior art computer architecture structure;

FIG. 2 depicts a block diagram of a data processing system in which the present invention may be implemented;

FIG. 3 depicts a block diagram of a personal digital assistant (PDA) in which the present invention may be implemented;

FIG. 4A depicts a block diagram illustrating a data structure for strongly encapsulating an application in accordance with the present invention;

FIG. 4B depicts a block diagram of a new model for a computer architecture in accordance with a preferred embodiment of the present invention;

FIG. 5 depicts a portion of XML code demonstrating one method the requirements part of the working definition may be represented in accordance with a preferred embodiment of the present invention;

FIG. 6 depicts a block diagram illustrating a method of automated damage detection and repair within a computing system in accordance with a preferred embodiment of the present invention; and

FIG. 7 depicts a flowchart illustrating an exemplary method of implementing a damage detection and repair facility in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

With reference now to the figures and, in particular, with reference to FIG. 1, a block diagram illustrating a basic prior art computer architecture structure is depicted. Before proceeding, it should be noted that throughout this description, identical reference numbers refer to similar or identical features in the different Figures.

All existing computer architectures are patterned after a core, basic computer architecture 100. This core, basic computer architecture is comprised of four elements: one or more processors 106, one or more memory spaces 104, one or more mechanisms for managing input/output 108, and one or more mechanisms for defining persistence 102. The Processor(s) 106 perform(s) the computations and instructions of a given application by loading its initial state into memory from the persisted image for that application.

Persistence 102 is the persisted storage of the applications apart from memory 104. The basic nature of computer systems requires that all applications be stored somewhere. Even if a person types a program into a computer system every time they wish to use that program, the program is always stored somewhere, even if that storage is in someone's head. How an application is stored does not vary much from platform to platform. Applications are stored as, for example, executable files and libraries, files, databases, registry entries, and environment variables. Typically these files, databases, etc. are stored on a physical nonvolatile storage device such as, for example, Read only Memory chips, a hard disk, a tape, CD-ROM, or a DVD. Even in the most complex applications, the number of distinct persisted structures is really rather small, although there can be many of each type.

Regardless of the application's complexity, it depends on its persisted image. If the persisted structure is correct, the

application will execute. If the persisted structure is wrong, the application will not execute.

With reference now to FIG. 2, a block diagram of a data processing system in which the present invention may be implemented is illustrated. Data processing system 250 is an example of a computer which conforms to the architecture depicted in FIG. 1. Data processing system 250 employs a peripheral component interconnect (PCI) local bus architecture. Although the depicted example employs a PCI bus, other bus architectures such as Micro Channel and ISA may be used. Processor 252 and main memory 254 are connected to PCI local bus 256 through PCI Bridge 258. PCI Bridge 258 also may include an integrated memory controller and cache memory for processor 252. Additional connections to PCI local bus 256 may be made through direct component interconnection or through add-in boards. In the depicted example, local area network (LAN) adapter 260, SCSI host bus adapter 262, and expansion bus interface 264 are connected to PCI local bus 256 by direct component connection. In contrast, audio adapter 266, graphics adapter 268, and audio/video adapter (A/V) 269 are connected to PCI local bus 266 by add-in boards inserted into expansion slots. Expansion bus interface 264 provides a connection for a keyboard and mouse adapter 270, modem 272, and additional memory 274. SCSI host bus adapter 262 provides a connection for hard disk drive 276, tape drive 278, and CD-ROM 280 in the depicted example. Typical PCI local bus implementations will support three or four PCI expansion slots or add-in connectors.

An operating system runs on processor 252 and is used to coordinate and provide control of various components within data processing system 250 in FIG. 2. The operating system may be a commercially available operating system such as JavaOS For Business or OS/2, which are available from International Business Machines Corporation. Those of ordinary skill in the art will appreciate that the hardware in FIG. 2 may vary depending on the implementation. For example, other peripheral devices, such as optical disk drives and the like may be used in addition to or in place of the hardware depicted in FIG. 2. The depicted example is not meant to imply architectural limitations with respect to the present invention. For example, the processes of the present invention may be applied to a multiprocessor data processing system.

Turning now to FIG. 3, a block diagram of a personal digital assistant (PDA) is illustrated in which the present invention may be implemented. A PDA is a data processing system (i.e., a computer) which is small and portable. As with the computer depicted in FIG. 2, and as with all computers, PDA 300 conforms to the computer architecture depicted in FIG. 1. The PDA is typically a palmtop computer, such as, for example, a Palm VII®, a product and registered trademark of 3Com Corporation in Santa Clara, Calif., which may be connected to a wireless communications network and which may provide voice, fax, e-mail, and/or other types of communication. The PDA 300 may perform other types of facilities to the user as well, such as, for example, provide a calendar and day planner. The PDA 300 may have one or more processors 302, such as a microprocessor, a main memory 304, a disk memory 306, and an I/O 308 such as a mouse, keyboard, or pen-type input, and a screen or monitor. The PDA 300 may also have a wireless transceiver 310 connected to an antenna 312 configured to transmit and receive wireless communications. The processor 302, memories 304, 306, I/O 308, and transceiver are connected to a bus 304. The bus transfers data, i.e., instructions and information, between each of the

devices connected to it. The I/O 308 may permit faxes, e-mail, or optical images to be displayed on a monitor or printed out by a printer. The I/O 308 may be connected to a microphone 316 and a speaker 318 so that voice or sound information may be sent and received.

The computers depicted in FIGS. 2 and 3 are examples of computers that conform to the computer architecture paradigm depicted in FIG. 1. The sophistication of the system, number of components, and speed of the processor may vary, but the basic model is the same as the architecture 100 depicted in FIG. 1: means for input and output such as a keyboard and display (whether LED or a video display terminal), a processor, memory, and persistence. The persistence may be, for example, stored on a hard disk or hard wired into the system. However, the computers depicted in FIGS. 2 and 3 are merely examples. Other computers, in fact all current computers, also conform to this model. Examples of other such computers include, but are not limited to, main frame computers, work stations, laptop computers, and game consoles, both portable and conventional game consoles that connect to a television. Examples of game consoles include, for example, a Sony Playstation® and a Nintendo Gameboy®. Playstation is a trademarked product of Sony Corporation of Tokyo, Japan. Gameboy is a product and a registered trademark of Nintendo of America, Inc. of Redmond, Wash., a wholly owned subsidiary of Nintendo Co., Ltd., of Kyoto, Japan.

The computer of FIG. 2 and the PDA of FIG. 3 are also suitable to implement the data structures, methods, and apparatus of the present invention. The present invention modifies the computer architecture paradigm illustrated in FIG. 1 in a way such as to present a new computer architecture paradigm with features and advantages as described below.

The present invention provides a data structure, process, system, and apparatus for allowing applications and computer systems to configure and manage themselves. The present invention also provides a process, system, and apparatus for creating encapsulated applications with controlled separation from an application's runtime representation. This separation is critical and necessary. The application's Encapsulation is provided by the Working Definition for the application, while the runtime image allows the application to execute within traditional execution environments, as defined by Windows, Unix, OS/2, and other operating system platforms.

With reference now to FIG. 4A, a block diagram illustrating a data structure for strongly encapsulating an application is depicted in accordance with the present invention. In a preferred embodiment, any software application may consist of the following elements: identity 404, code 406, requirements 408, data 410, artifacts 412, and settings 414. These elements structure the application's defining characteristics. The defining characteristics define the persisted state, settings, and structures required to build a valid runtime representation of the application within its targeted computer system. The application's working definition documents and controls each of these elements of the application.

The application's identity 404 is defined by the application developer, and consists of the application's name, version, etc. The identity section 404 may also provide documentation, web links, etc. for the application useful both to automated management services and users.

The code 406 represents the executable images, files, source code, and any other representation of the executable

portion of the application that the application developer may decide to use. The code section 406 may also include executable images, files, source code, etc. to enable the installation, maintenance, configuration, uninstall, etc. of the application through the application's life cycle.

The requirements section 408 defines what directory structures, environment variables, registry settings, and other persisted structures must exist and in what form for this application to be properly constructed and installed in the persisted image of a computer system. The requirements section 408 also details any services and applications that are required to support this application. The requirements section 408 details these concepts as a set of constraints, and also may define the order in which these constraints must be resolved when such order is required. The requirements 408 also define the circumstances and constraints for the use of application specific installation, maintenance, configuration, uninstall, etc. code.

The Data section 410 defines data tables, configuration files, and other persisted structures for the application. The data section 410 is used to hold UI preferences, routing tables, and other information that makes the application more usable.

The Artifacts 412 are the persisted form of the value provided by the application. Examples of Artifacts 412 include documents, spreadsheets, databases, mail folders, text files, image files, web pages, etc. These are the files that contain the user's work ("user" in this context can be human or application).

The settings 414 are the persisted structures created or modified within the runtime representation that are intended to satisfy the requirements for the application. A distinction is drawn between the requirements 408 (which may specify a range of values in a registry setting or the form a directory structure must conform to) and the actual solution constructed in the runtime image 100 to satisfy these requirements.

The Requirements 408 are provided by the developers, application management software, environments, etc. Settings are made in an effort to resolve those requirements, and identify what requirements a setting is intended to satisfy.

With reference now to FIG. 4B, a block diagram of a new model for a computer architecture 400 is depicted in accordance with a preferred embodiment of the present invention. The new computer architecture 400 of the present invention builds on the standard architecture as depicted in FIG. 1. However, the new computer architecture 400 also includes an application's working definition 402. In a preferred embodiment, an application's Working definition 402 is an extensible markup language (XML) representation of the identity 404, code 406, requirements 408, data 410, artifacts 412, and settings 414; that is it includes all of the defining characteristics for the application. By defining these elements, separate from the runtime representation 100 of an application, working definitions 402 provides a way to strongly encapsulate an application which is compatible with operating systems that expect applications in an unencapsulated form. Working definitions define a new, universal picture of a computer system. Working definitions do not interfere with any operating system or platform because they have no active behavior. Working definitions define the application's runtime representation 100 by defining what an application is, what it requires, and what was done to give it what it needs, within a given computer system. While working definitions have no active behavior, they enable the implementation of a variety of automated services. These

services include application life cycle management, resource tracking, application installation, damage detection and repair, computer system optimization, etc.

Strong encapsulation of state greatly reduces the global complexity of a computer system. States such as, for example, path requirements, file extensions, registry settings, program files, can be maintained in a pure state outside of the runtime representation 100. Because the runtime version of the application 100 is a redundant representation, it can be reconstructed, and fixed using the encapsulated version of the application as the standard. Important modifications can be persisted back to the application's Working Definition as a background task.

Working definitions use XML to define a flexible format for structuring the critical information about an application, thus encapsulating the application. XML provides the ability to extend working definitions to fit future requirements and, XML, just like the hypertext markup language (HTML) of the world wide web, can be as simple or as complex as the job requires. The actual size of a working definition is very small compared to the application it defines, even if the application is just a batch file. At the same time, working definitions can be used to describe vast, distributed, multi-platform applications.

Working definitions are platform and technology independent and address universal configuration issues. Furthermore, working definitions can be used as an open standard. Working definitions provide configuration information to services and to applications as well as providing bookkeeping support.

An application's valid runtime representation is one that satisfies a finite, structured set of constraints. Therefore, XML is an ideal method because XML provides an ideal representation for structured data and supports the ability to define links that allow tracking of internal relationships or for including references to structures defined externally. Furthermore, XML is designed to be easily extended and is platform independent.

However, XML is not a good representation for constant modification and access. The runtime representation for the application provides the application with an efficient executable representation. To provide an application with an efficient executable representation, the XML can detail the required runtime structure and requirements for the application. Using these instructions, the application can be constructed and executed automatically. Prerequisites, in the proper order and with the proper configuration will also be constructed and executed automatically.

As the application executes, changes made to the application's state may be made as they are presently, that is, to the runtime representation. The XML specifies the important files to be persisted, and this persistence may be done in the background. The overhead is thus very minimal with little effect on the performance as viewed from a user's perspective.

The nature of the problem of automating management and configuration of computer systems requires a common, computer friendly and people friendly, complete representation of each software component. Without this information, the automated management of computer systems is not possible.

While the working definitions have been described herein and will be described henceforth with reference to an XML representation of the application, any number of other technologies and approaches might be used as well as will be obvious to one of ordinary skilled in the art. For example,

encapsulation could also be implemented using databases or with Zip files. Also, a database could be used for the actual implementation of the support for the encapsulation with XML used as a transfer protocol.

Furthermore, although described primarily with reference to constructing Working Definitions using XML, Working Definitions may be constructed using any structured format. Possible choices, for example, include the use of other structured forms. These include:

Object based technologies (Java Beans, CORBA objects, C++ objects, etc.). These would have to be backed by some sort of database.

Object Oriented databases. These are a mix of databases and Object based technologies.

Functional technologies. One could build a PostScript-like or Lisp-like language (compiled or interpreted) that defines these structures.

Rule based technologies. Since Working Definitions generally resolve sets of constraints, this may well be the best way of implementing services that execution against the constraints that Working Definitions define. Working definitions could be constructed directly into Rule form, similar to forward chaining technologies such as ART, or backward chaining technologies such as Prolog.

Tagged formats. XML is an example of a tagged format. However, there are other tagged formats that would work just as well. Tagged Image Format (TIFF), although generally used for defining images, may also be used to define Working Definitions since the use of custom tags is supported. Additionally, since TIFF is a binary format, it could hold the executable code and binary data. Other tagged formats include SGML, TeX, and LaTeX.

In addition to these examples of structured formats, there are certainly other formats and representations that would also work for constructing Working Definitions. Any technology that can 1) represent structured data, 2) support links to outside sources, and 3) can be used across platforms could be used to define Working Definitions.

With reference now to FIG. 5, a portion of XML code is shown, demonstrating one method for representing the requirements element of the working definition, in accordance with a preferred embodiment of the present invention. The portion 500 of XML depicted is only snippet of a representation of the requirements element of the working definition and is given merely as an example. Furthermore, this portion 500 of XML is overly simplified, but does demonstrate how this information might be represented.

Platform tag at 502 reflects the requirement of the application that the platform be an xx86 type of computer, such as, for example, an Intel Pentium® class computer, running a Windows 95™, Windows 98™, or Windows NT™ operating system. Pentium is a registered trademark of Intel Corporation of Santa Clara, Calif. Windows 95™, Windows 98™, and Windows NT™ are all either trademarks or registered trademarks of Microsoft Corporation of Redmond, Wash.

Services tag at 504 indicates that the application requires the TCP/IP and file system services. Prerequisites tag at 506 indicates that the application Adobe Acrobat® is a prerequisite (i.e., requirement) for this application. Adobe Acrobat is a registered trademark of Adobe Systems Incorporated of San Jose, Calif.

The registry tag at 508 describes a registry entry. In this example, a Registry entry and a couple of keys are defined.

The Directory tag at 510 describes a couple of directories, the "programRoot" (which is required, but not specified by name) and a subdirectory "programRoot"/jclass.

Section 512 describes adding the programRoot directory to the path environment variable. The directory "programRoot"/jclass is also described as necessary to be added to the classpath environment variable.

The <test>...</test>-tags in Section 512 describe tests that can be used to verify the setup of the enclosing section. This is one possible way of describing how to use application specific code to maintain an application.

With reference now to FIG. 6, a block diagram illustrating a method of automated damage detection and repair within a computing system is depicted in accordance with a preferred embodiment of the present invention. Currently, in the prior art, each application is responsible for its own integrity within a computer system. Automated facilities for detecting conflicts and resolving them, detecting damaged files or detecting missing registry entries are severely limited with prior art systems.

Conflicts and modifications to the runtime environment 100 cannot be avoided. They are the natural result of using a computer system, since the use of a computer system naturally leads to installing new applications, application updates, production and management of application artifacts, use of temporary files, and other changes in the system's configuration.

Damage detection and repair facility 602 uses the requirements defined in the working definitions 604 of all the installed applications as a set of constraints. Whenever changes occur to the runtime environment 300 or to any of the encapsulated applications, damage detection and repair facility 602 detects these changes and checks them against the set of constraints defined by the working definitions of the encapsulated applications installed on the computer system. The XML working definitions of each encapsulated application define the invariant portions of the application and the user's work which should be persisted as contrasted with temporary files that need not be persisted. An additional digitally signed section of the XML application working definition can provide insurance that checksums and file sizes for each of the application's files in the runtime environment 100 have not been modified. If a file has been modified, as detected by damage detection and repair facility 602, that file can be repaired with a signed, known valid version.

The set of files subject to inspection is very limited, since the working definitions define exactly which files should be checked. Instead of having to verify each and every file, the search can be limited to only those files the working definitions identify as critical. More exhaustive checks can be done, but the basic runtime representation verification does not require them.

With reference now to FIG. 7, a flowchart illustrating an exemplary method of implementing a damage detection and repair facility, such as, for example damage detection and repair facility 602 in FIG. 6, is depicted in accordance with the present invention.

The damage detection and repair facility monitors the computer system for changes (step 702) and determines whether the data processing system has received an indication to power down (step 703). If an indication to power down the data processing system has been received, then the process ends. If no indication to power down has been received, then the damage detection and repair facility determines whether changes have been made to any files, settings or encapsulated applications (step 704). Searches

can be avoided when applications use a notification facility in order to facilitate this process. However, this kind of interaction with the application is not required. If no changes have been made, then the damage detection and repair facility 602 continues to monitor the computer system (step 702).

If changes have been made to some aspect of the runtime representation of an application as defined by that application's working definition, then the damage detection and repair facility compares the change against the constraints as defined by the set of working definitions in the computer system (step 706) and determines whether the change creates any conflict (step 708). If the change effects one or more working definitions without conflict, it is recorded in the settings section for the effected application(s) (step 710). Conflicts are resolved by restoring or adjusting the runtime representations effected. Changes to temporary files and settings (as defined by the working definitions of the applications that use them) do not cause conflicts.

In systems where security is very important, an optional test (step 711) for changes outside those defined as reasonable can be made. In this optional version, if the change does not create any conflict, then the damage detection and repair facility determines whether the change effects any working definition defined software component (step 711). This is done to detect those changes outside the defined constraints of the system. If the change does effect a working definition software component, then the settings of the effected working definitions are updated (step 712). If the change does not effect any working definition defined software component, then any number of strategies may be used (step 713) to address this security concern, including reversing the changes, logging the changes for inspection later, or reporting these changes to some monitoring facility. The damage detection and repair facility then continues to monitor the system (step 702).

Thus, the working definitions of installed applications allow a computer system to be modeled as a set of applications that impose a set of constraints on the runtime representation of the computer system. These constraints define all of the elements of an application that can possibly be persisted as described above. When settings change, the damage detection and repair facility can evaluate these changes against this set of constraints as defined by the working definitions of each application. If the constraints are still met, then the settings can be recorded in the settings section of the effected working definitions. If the constraints are not met, the settings can be adjusted (and recorded) as required to restore the computer system back to its proper configuration.

It is important to note that while working definitions embody logically the totality of all of the defining characteristics of an application, the working definition may in fact be implemented as a distributed entity, rather than the single entity located on an individual machine as the present invention has been primarily described. Components of it may be accessed via links and may be physically stored on servers, hard drives, or other media and accessed over buses, the Internet, an intranet, or other channels.

It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in the form of a computer readable medium of instructions and a variety of forms and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry

11

out the distribution. Examples of computer readable media include recordable-type media, such as a floppy disk, a hard disk drive, a RAM, CD-ROMs, DVD-ROMs, and transmission-type media, such as digital and analog communications links, wired or wireless communications links using transmission forms, such as, for example, radio frequency and light wave transmissions. The computer readable media may take the form of coded formats that are decoded for actual use in a particular data processing system.

The description of the present invention has been presented for purposes of illustration and description, and is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiment was chosen and described in order to best explain the principles of the invention, the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

What is claimed is:

1. A method of detecting and repairing damaged portions of a computer system, the method comprising:
 - detecting a change to the computer system;
 - comparing the change to working definitions for each application installed on the computer system; wherein the working definitions comprise a set of constraints placed upon the computer system by each application installed on the computer system; and
 - responsive to a determination that the change is in conflict with the set of constraints, modifying a persistent storage so as to resolve the conflict.
2. The method as recited in claim 1, further comprising: responsive to a determination that the change does not create a conflict with the set of constraints, updating settings in the working definitions of effected applications.
3. The method as recited in claim 1, wherein the working definition of each application is provided by an extensible markup language representation.
4. The method as recited in claim 1, wherein the step of modifying the persistent storage comprises repairing the runtime image of an application based on an encapsulated representation of the application.
5. The method as recited in claim 1, wherein the step of modifying the persistent storage comprises repairing a damaged file using a correct version of the file from the working definition.
6. The method as recited in claim 5, wherein the correct version of the file is retrieved from a server.
7. The method as recited in claim 6, wherein the server is accessed via a network.
8. The method as recited in claim 7, wherein the network is an Internet.
9. The method as recited in claim 1, further comprising:
 - responsive to a determination that the change does not create a conflict with the set of constraints, determining whether the change effects any working definition defined software component; and
 - responsive to a determination that the change does not effect any working definition defined software component, reversing the change.
10. The method as recited in claim 1, further comprising:
 - responsive to a determination that the change does not create a conflict with the set of constraints, determining whether the change effects any working definition defined software component; and

12

responsive to a determination that the change does not effect any working definition defined software component, logging the change.

11. The method as recited in claim 1, further comprising:
 - responsive to a determination that the change does not create a conflict with the set of constraints, determining whether the change effects any working definition defined software component; and
 - responsive to a determination that the change does not effect any working definition defined software component, reporting the change to a monitoring facility.
12. A computer program product in computer readable media for use in a data processing system for detecting and repairing damaged portions of a computer system, the computer program product comprising:
 - first instructions for detecting a change to the computer system;
 - second instructions for comparing the change to working definitions for each application installed on the computer system; wherein the working definitions comprise a set of constraints placed upon the computer system by each application installed on the computer system; and
 - third instructions, responsive to a determination that the change is in conflict with the set of constraints, for modifying a persistent storage so as to resolve the conflict.
13. The computer program product as recited in claim 12, further comprising:
 - fourth instructions, responsive to a determination that the change does not create a conflict with the set of constraints, for updating settings in the working definitions of effected applications.
14. The computer program product as recited in claim 12, wherein the working definition of each application is provided by an extensible markup language representation.
15. The computer program product as recited in claim 12, wherein the step of modifying the persistent storage comprises repairing the runtime image of an application based on an encapsulated representation of the application.
16. The computer program product as recited in claim 12, wherein the step of modifying the persistent storage comprises repairing a damaged file using a correct version of the file from the working definition.
17. The computer program product as recited in claim 16, wherein the correct version of the file is retrieved from a server.
18. The computer program product as recited in claim 17, wherein the server is accessed via a network.
19. The computer program product as recited in claim 18, wherein the network is an Internet.
20. The computer program product as recited in claim 12, further comprising:
 - fourth instructions, responsive to a determination that the change does not create a conflict with the set of constraints, for determining whether the change effects any working definition defined software component; and
 - fifth instructions, responsive to a determination that the change does not effect any working definition defined software component, for reversing the change.
21. The computer program product as recited in claim 12, further comprising:
 - fourth instructions, responsive to a determination that the change does not create a conflict with the set of constraints, for determining whether the change effects any working definition defined software component; and

13

fifth instructions, responsive to a determination that the change does not effect any working definition defined software component, for logging the change.

22. The computer program product as recited in claim 12, further comprising:

fourth instructions, responsive to a determination that the change does not create a conflict with the set of constraints, for determining whether the change effects any working definition defined software component; and

fifth instructions, responsive to a determination that the change does not effect any working definition defined software component, for reporting the change to a monitoring facility.

23. A system for detecting and repairing damaged portions of a computer system, the system comprising:

means for detecting a change to the computer system;

means for comparing the change to working definitions for each application installed on the computer system; wherein the working definitions comprise a set of constraints placed upon the computer system by each application installed on the computer system; and

means, responsive to a determination that the change is in conflict with the set of constraints, for modifying a persistent storage so as to resolve the conflict.

24. The system as recited in claim 23, further comprising: means, responsive to a determination that the change does not create a conflict with the set of constraints, for updating settings in the working definitions of effected applications.

25. The system as recited in claim 23, wherein the working definition of each application is provided by an extensible markup language representation.

26. The system as recited in claim 23, wherein the step of modifying the persistent storage comprises repairing the runtime image of an application based on an encapsulated representation of the application.

14

27. The system as recited in claim 23, wherein the step of modifying the persistent storage comprises repairing a damaged file using a correct version of the file from the working definition.

28. The system as recited in claim 27, wherein the correct version of the file is retrieved from a server.

29. The system as recited in claim 28, wherein the server is accessed via a network.

30. The system as recited in claim 29, wherein the network is an Internet.

31. The system as recited in claim 23, further comprising:

means, responsive to a determination that the change does not create a conflict with the set of constraints, for determining whether the change effects any working definition defined software component; and

means, responsive to a determination that the change does not effect any working definition defined software component, for reversing the change.

32. The system as recited in claim 23, further comprising:

means, responsive to a determination that the change does not create a conflict with the set of constraints, for determining whether the change effects any working definition defined software component; and

means, responsive to a determination that the change does not effect any working definition defined software component, for logging the change.

33. The system as recited in claim 23, further comprising:

means, responsive to a determination that the change does not create a conflict with the set of constraints, determining whether the change effects any working definition defined software component; and

means, responsive to a determination that the change does not effect any working definition defined software component, for reporting the change to a monitoring facility.

* * * * *



US006571389B1

(12) **United States Patent**
Spyker et al.

(10) **Patent No.: US 6,571,389 B1**
 (45) **Date of Patent: May 27, 2003**

(54) **SYSTEM AND METHOD FOR IMPROVING THE MANAGEABILITY AND USABILITY OF A JAVA ENVIRONMENT**

6,353,926 B1 * 3/2002 Parthesarathy et al. 717/168
 6,357,019 B1 * 3/2002 Blaisdell et al. 714/38
 6,381,742 B2 * 4/2002 Forbes et al. 717/178

* cited by examiner

(75) **Inventors:** Andrew W. Spyker, Raleigh, NC (US);
 Matthew David Walnock, Cary, NC (US)

Primary Examiner—Gregory Morse
Assistant Examiner—Chuck Kendall

(73) **Assignee:** International Business Machines Corporation, Armonk, NY (US)

(74) **Attorney, Agent, or Firm**—Jeanine S. Ray-Yarlettis; Marcia L. Doubet

(*) **Notice:** Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(57) **ABSTRACT**

(21) **Appl. No.:** 09/300,041

(22) **Filed:** Apr. 27, 1999

(51) **Int. Cl.**⁷ G06F 9/44

(52) **U.S. Cl.** 717/176; 717/176; 709/106; 709/242

(58) **Field of Search** 711/11; 709/204, 709/106, 242; 717/168–178

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,732,275 A * 3/1998 Kullick et al. 717/11
 5,752,042 A * 5/1998 Cole et al. 717/176
 5,764,992 A * 6/1998 Kullick et al. 717/11
 5,966,540 A * 10/1999 Lister et al. 717/178
 5,983,348 A * 11/1999 Ji 713/200
 5,995,756 A * 11/1999 Herrmann 395/712
 6,009,274 A * 12/1999 Fletcher et al. 395/712

A method, system, and computer-readable code for improving the manageability and usability of a Java environment. The advantages of applets and applications are combined, while avoiding particular disadvantages of both, resulting in a technique whereby all Java programs are executed without relying on use of a browser to provide a run-time environment. Techniques for improving the packaging of Java components, including run-time environments and extensions as well as applications, are defined. Dependencies are specified in a manner which enables them to be dynamically located and installed; and enables sharing dependent modules (including runtime environments) among applications. The dependency specification technique ensures that all dependent code will be automatically available at run-time, without requiring a user to perform manual installation. The run-time environment required for an application is specified, and a technique is provided for dynamically changing the runtime that will be used (including the ability to change run-times on a per-program basis), without requiring user intervention.

33 Claims, 11 Drawing Sheets

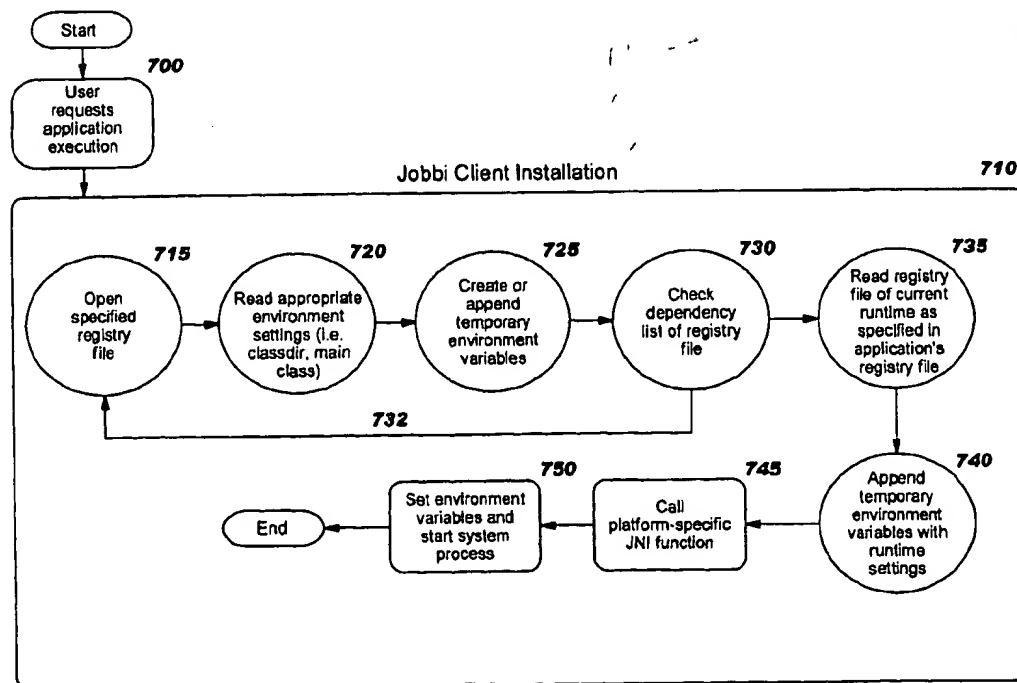


FIG. 1

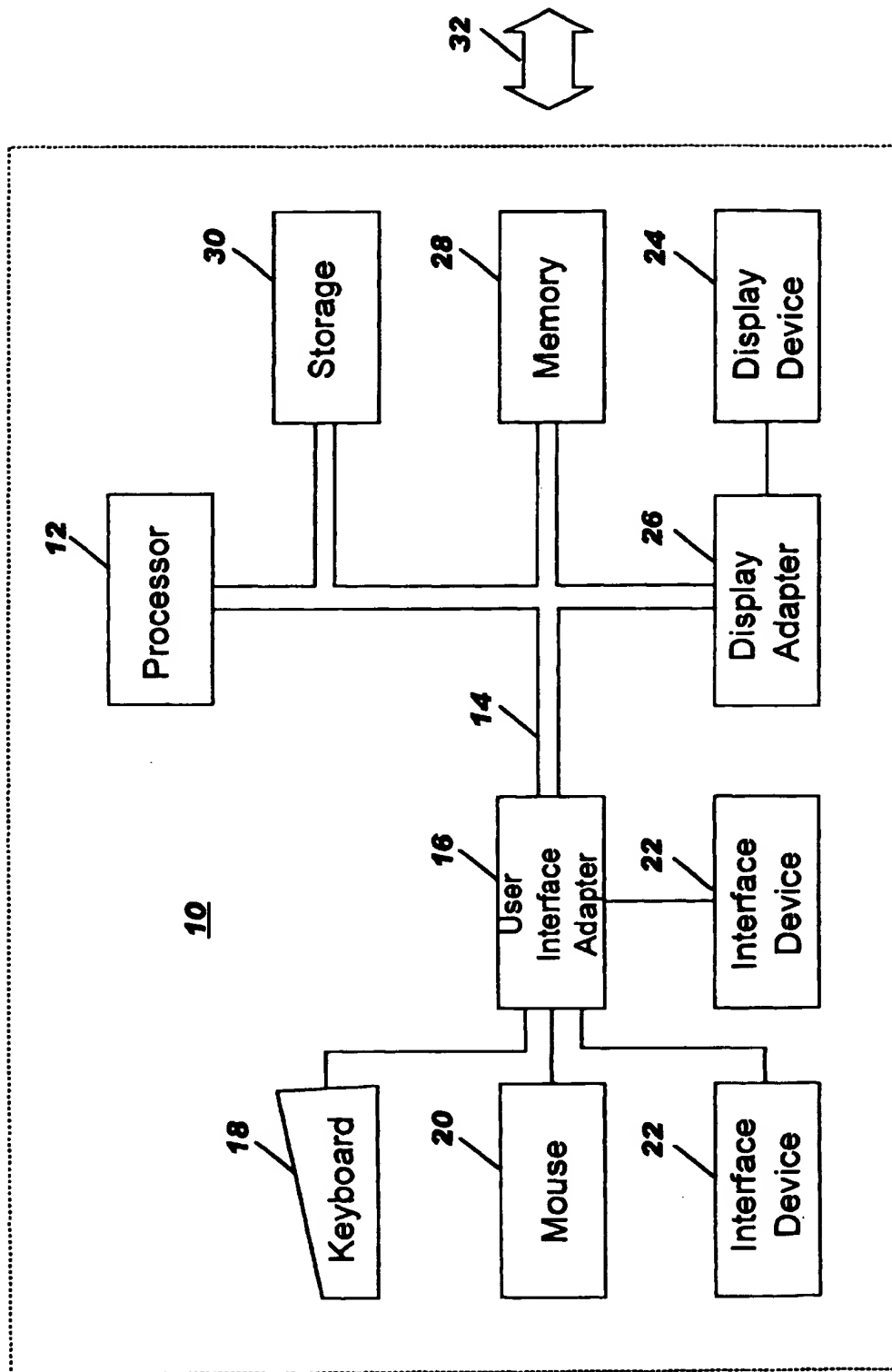


FIG. 2
(Prior Art)

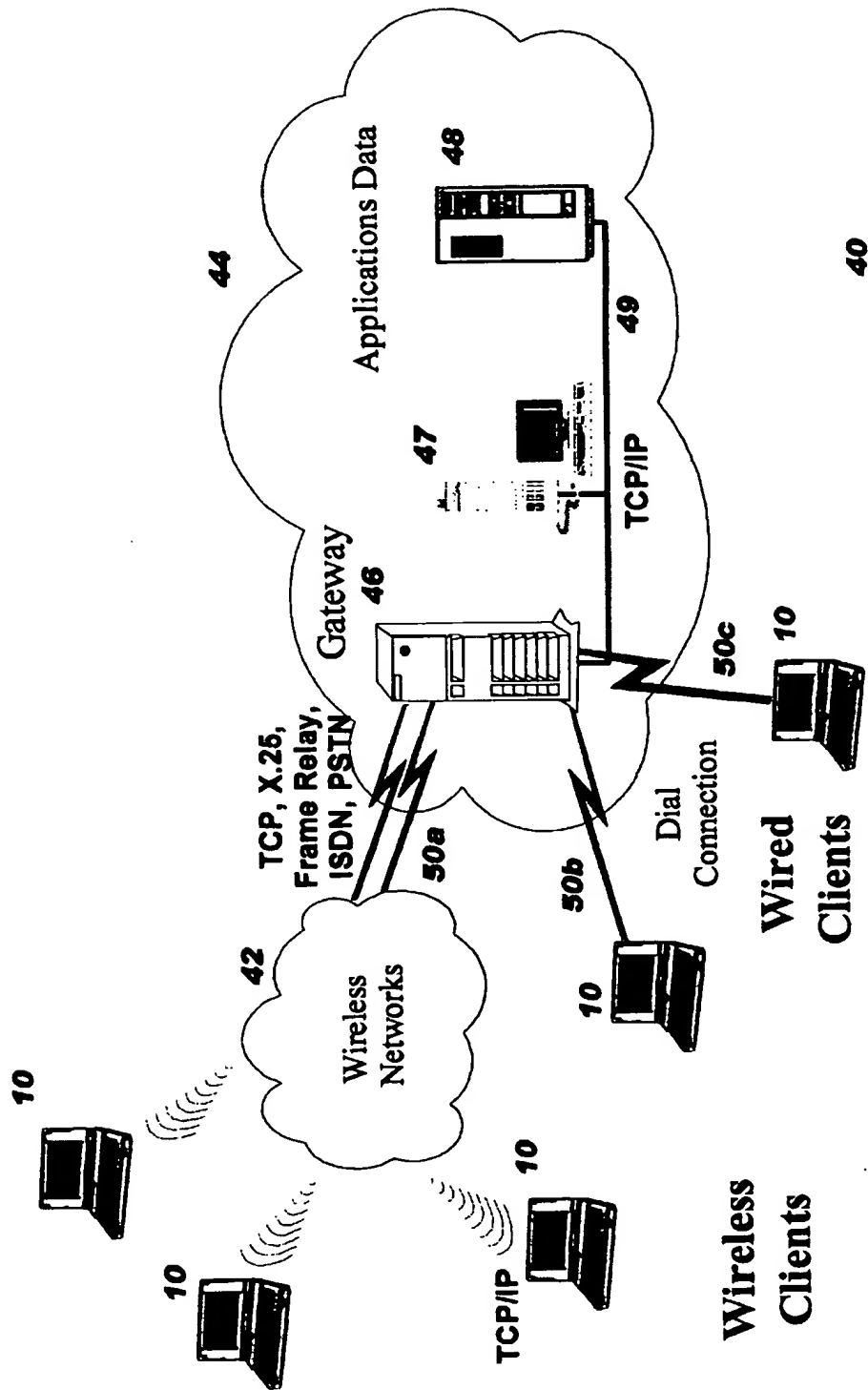


FIG. 3

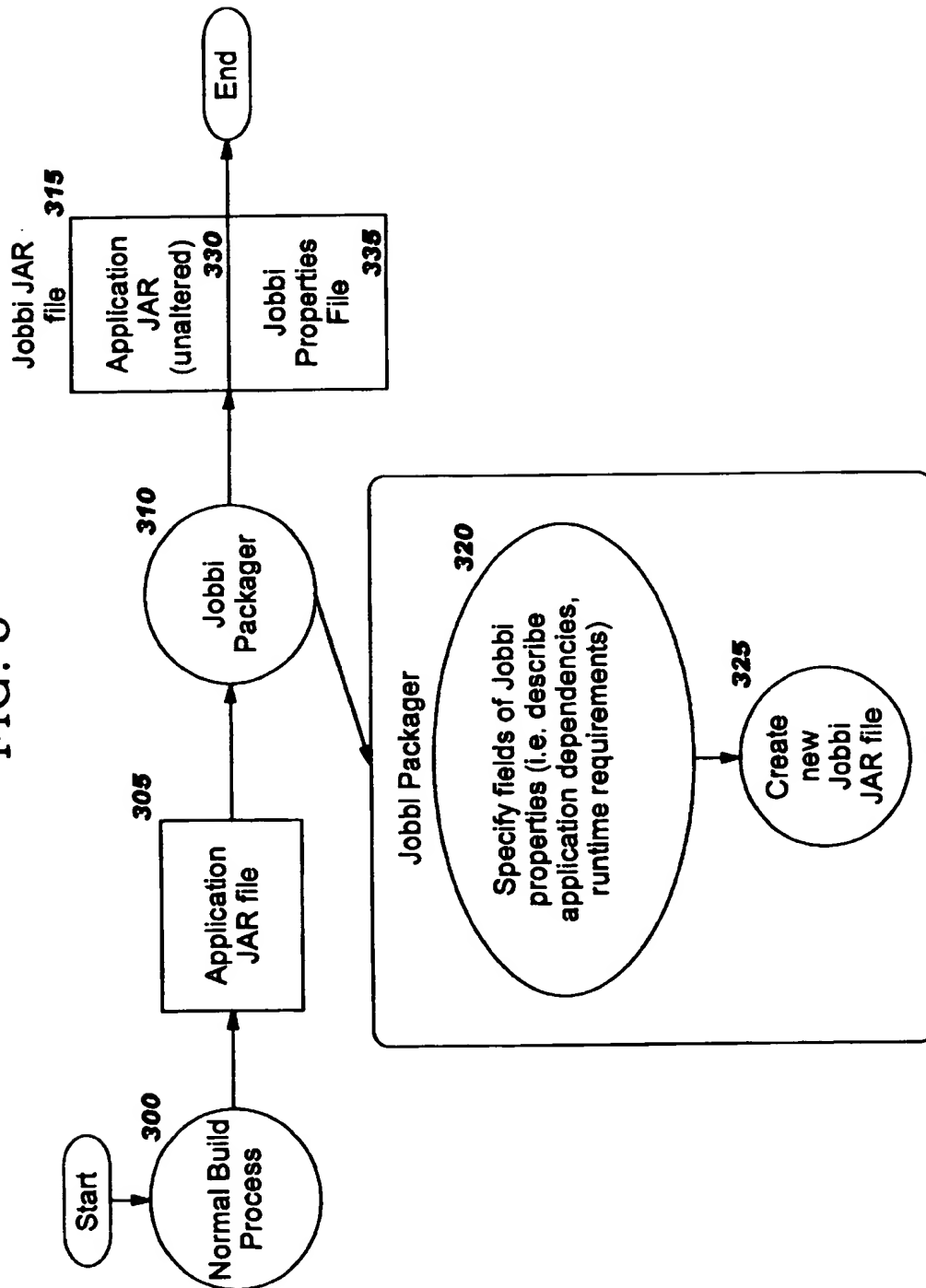


FIG. 4A

400

405 Property Key - displayname
Displayname

408 Property Key - displayicon
Display icon

410 Property Key - version
Comma separated list in form of Major.Minor.Rev.Build version information

415 Property Key - jobbitype
Either application, runtime, or extension
application - Can be added to the jobbi desktop
runtime - Can be added to the list of runtimes
extension - Can be added to the list of extensions

420 Property Key - jobbilocationtype
Either URL | file | jobbi-lookup-server
URL - the jobbi archive is located at the URL specified in jobbilocation
file - the jobbi archive is contained in a file name specified in jobbilocation
jobbi-lookup-server - goes to the jobbi-server and looks up the location information

425 Property Key - jobbilocation
Either a URL, filename, prompt, or '.'
URL - URL pointing to jobbi archive
filename - presents the user with a file chooser box specifying the name of the jobbi archive
428 prompt - presents the user with a dialog that either lets the user put in a URL or filename
'.' - use of this syntax indicates that the jobbi archive is contained within the current archive

430 Property Key - nativecode
Either true or false
true - jobbi archive contains native code
false - jobbi archive does not contain native code

435 Property Key - nativecodeplatform
One of the strings of the jobbi supported platforms
Empty if nativecode is false

440 Property Key - dependencies
Comma separated list of UID's that this package depends on
Empty if there are no dependencies for this jobbi package

445 Property Key - main
Java class name of class containing the main() function. This field is only specified if
jobbitype = application

FIG. 4B

Example
450

```

#X1's entry 455
#_____
displayname=x1's display name 459
displayicon=images\hod.gif 460
version=1,0,0 461
jobbitype=application 462
locationtype=file 463
location= 464
nativecode=false 465
nativecodeplatform= 466
dependencies=X2,Y1 467
main=com.ibm.X1 468

#X2's entry... 470
...

#X3's entry... 475
...

#Y1's entry 480
#_____
displayname=y1's display name 481
displayicon=
version=1,0,0 482
jobbitype=runtime 483
locationtype=URL 484
location=http://myserver.ibm.com/runtimey1.jar 485
nativecode=true 486
nativecodeplatform=Win32 487
dependencies= 488
main=

y2_...
...
y3_...
...

```

FIG. 5A

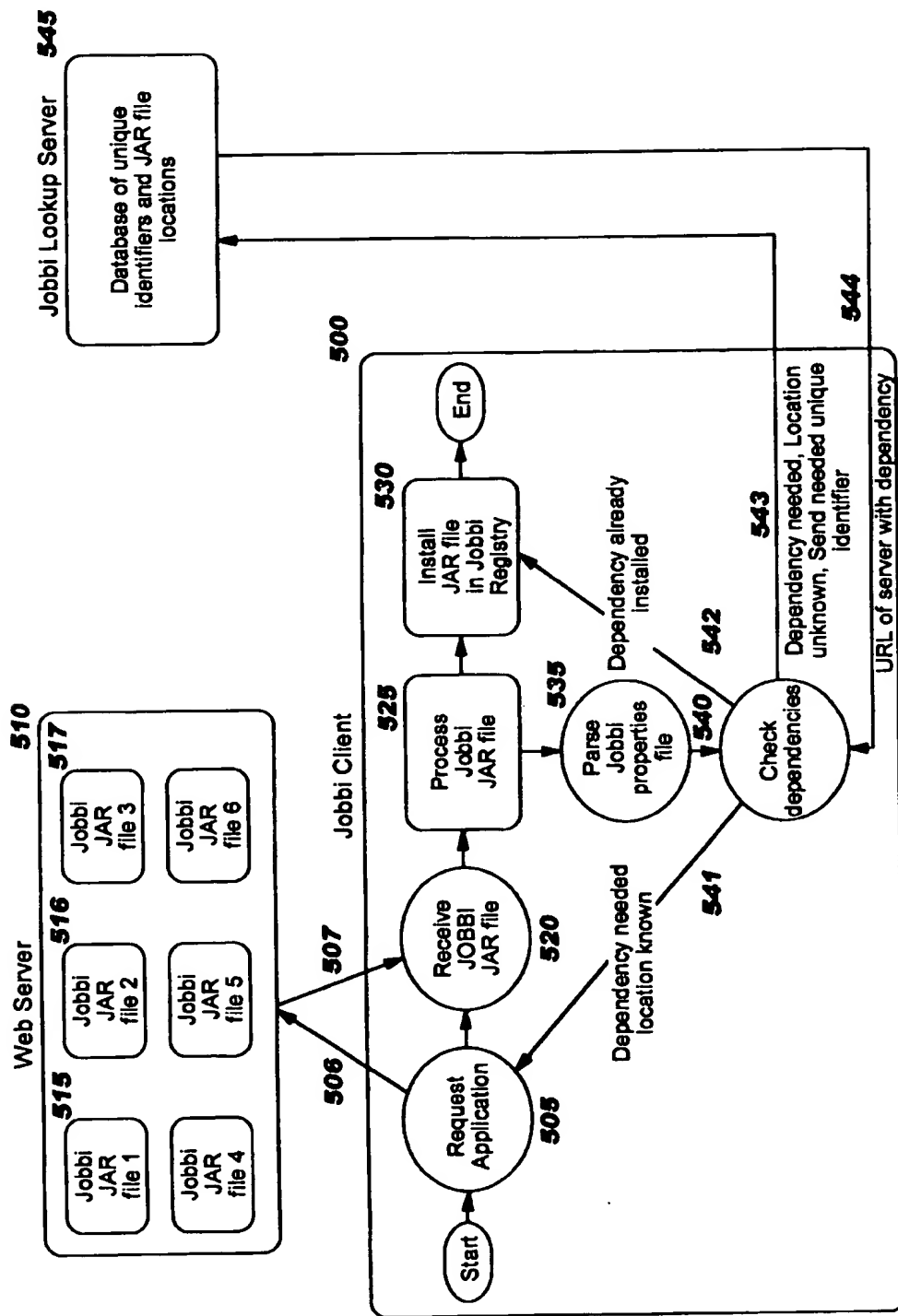


FIG. 5B

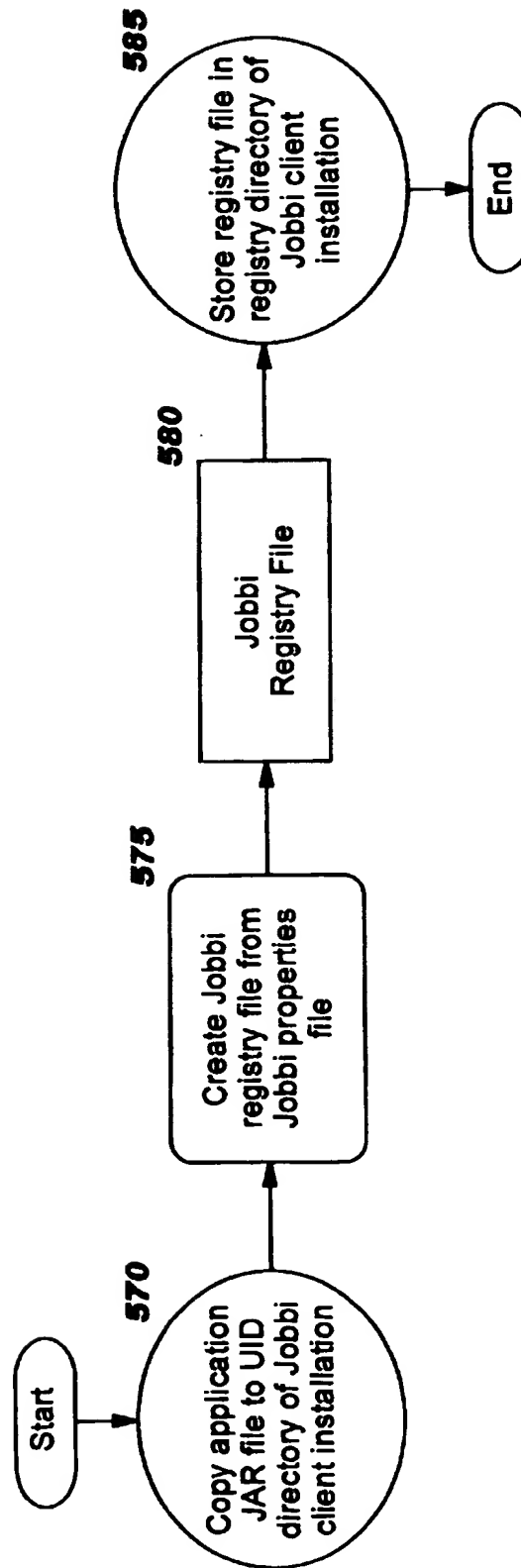


FIG. 6A

File format 600

Key	Value
605 jobbi.uid	Unique identifier of this package. This can be any valid string
610 jobbi.main	Java class containing the main() function. This is only specified when jobbi.type = 0 (application)
615 jobbi.displayname	Displayable string
620 jobbi.displayicon	Display icon path information
625 jobbi.classdir	relative directory or archive name which needs to be included in the classpath when this package is used.
630 jobbi.runtimelist	List of unique identifiers of runtimes in which this package can be executed
635 jobbi.working	The relative working directory which needs to be set in order to use this package
640 jobbi.type	0 = application 1 = extension 2 = runtime
645 jobbi.currentruntime	The unique identifier of the runtime which is to be used when this package is executed. This is only set when jobbi.type = 0
650 jobbi.package	The archive name of this package. This is only specified when the package archive is not expanded into the UID directory of the Jobbi client installation.
655 jobbi.extern	semi-colon separated list of unique identifiers on which this package is dependent
660 jobbi.param	semi-colon separated list of parameters which are to be passed to the main function of the application

The '|' character is used as a platform independent path separator.
The '&' character is used as a platform independent symbol for concatenation

FIG. 6B

Example 670

```
#Jobbi registry file for Host on Demand 3.0
#Thu Jan 21 23:19:26 EST 1999
jobbi.uid=00000001 671
jobbi.main=com.ibm.eNetwork.HOD.HostOnDemand 672
jobbi.displayname=Host on Demand 3.0 673
jobbi.displayicon=images\hod.gif 674
jobbi.classdir=hod30dbg.zip& 675
jobbi.runtimelist=10000001,10000002,10000004 676
jobbi.working=. 677
jobbi.type=0 678
jobbi.currentruntime=10000004 679
jobbi.package= 680
```

FIG. 7

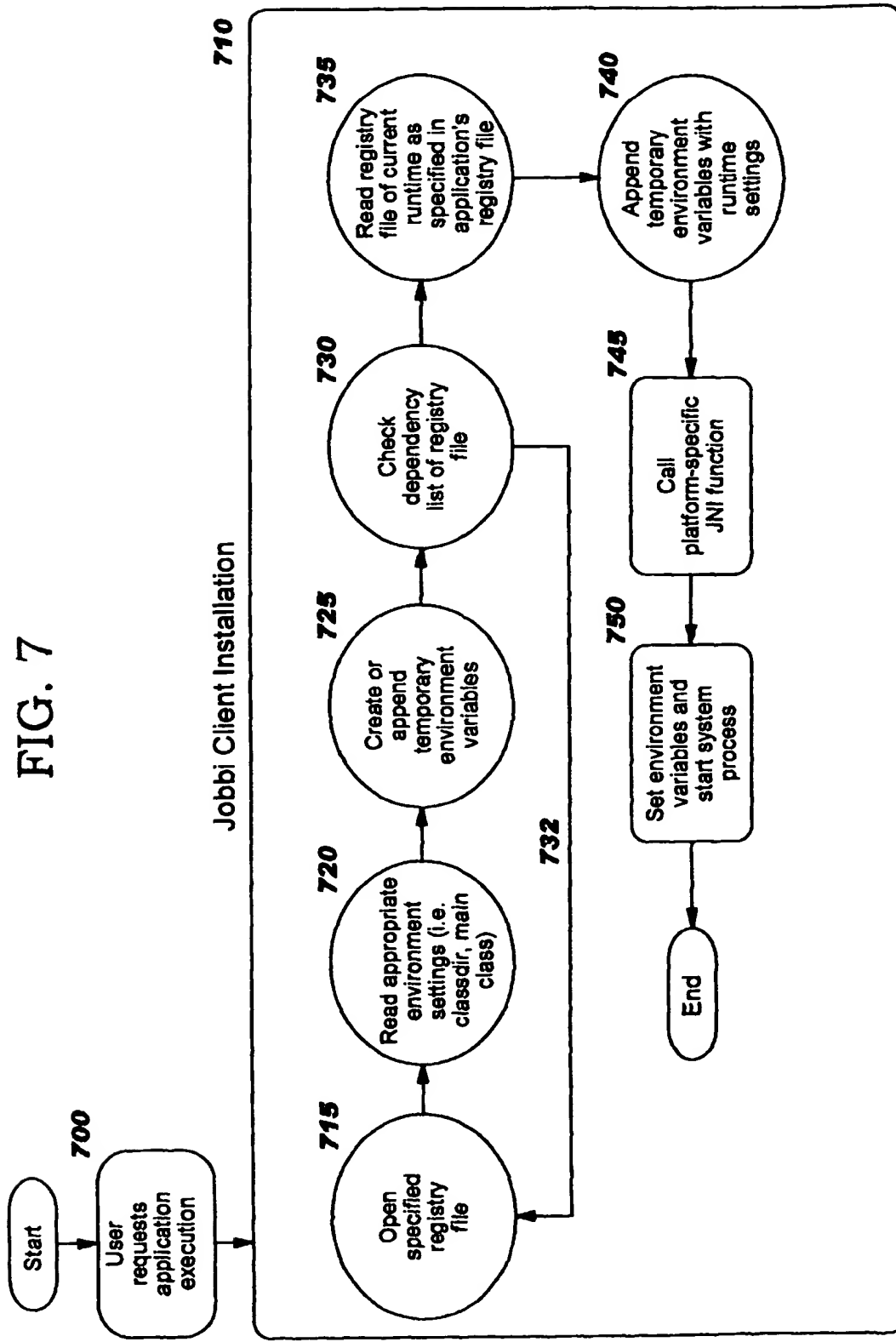
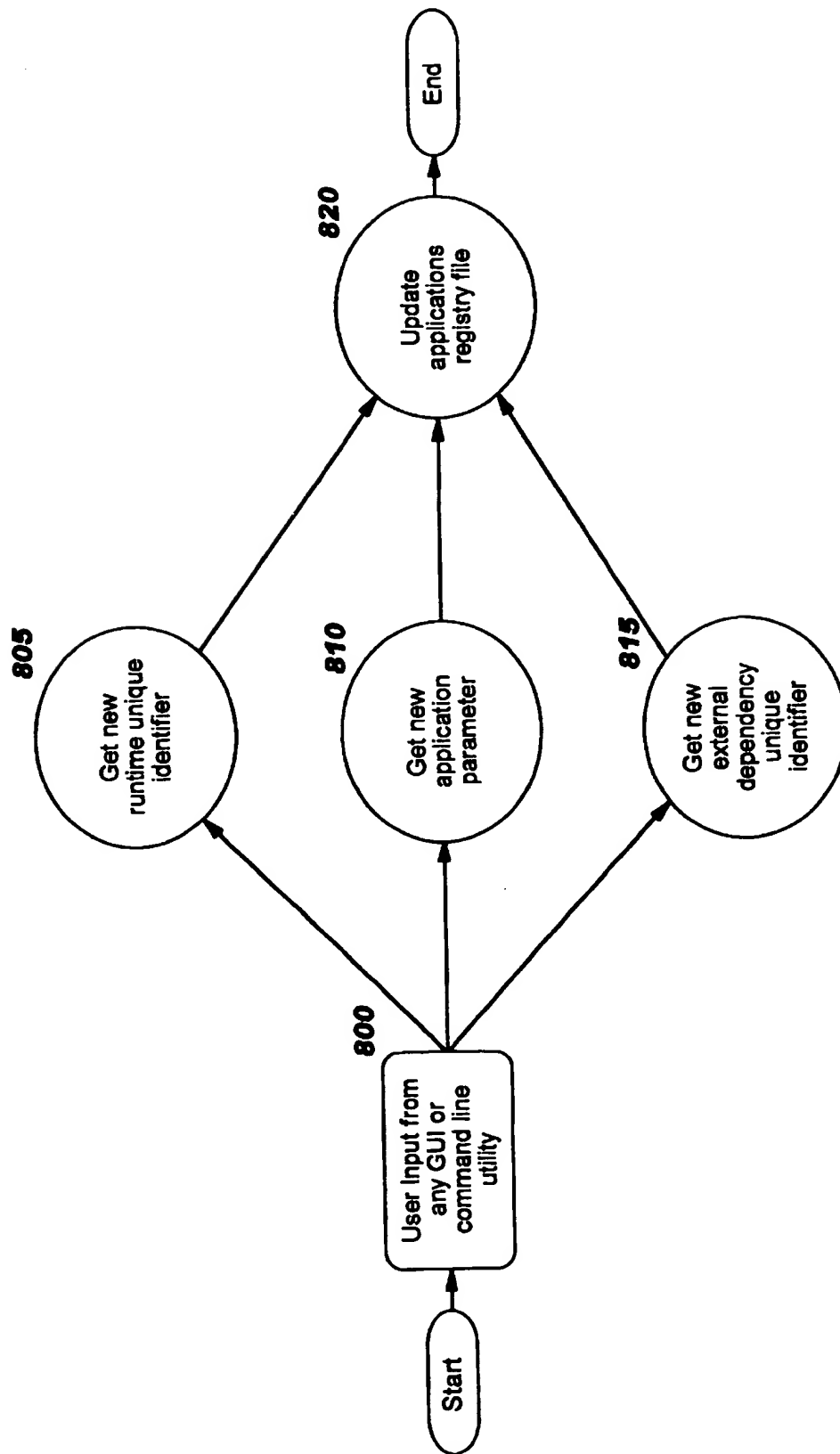


FIG. 8



SYSTEM AND METHOD FOR IMPROVING THE MANAGEABILITY AND USABILITY OF A JAVA ENVIRONMENT

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a computer system, and deals more particularly with a method, system, and computer-readable code for improving the manageability and usability of a Java environment.

2. Description of the Related Art

Java is a robust, portable object-oriented programming language developed by Sun Microsystems, Inc., and which is gaining wide acceptance for writing code for the Internet and World Wide Web. While compilers for most programming languages generate code for a particular operating environment, Java enables writing programs using a "write once, run anywhere" paradigm. ("Java" and "Write Once, Run Anywhere" are trademarks of Sun Microsystems, Inc.)

Java attains its portability through use of a specially-designed virtual machine ("VM"). This virtual machine is also referred to as a "Java Virtual Machine", or "JVM". The virtual machine enables isolating the details of the underlying hardware from the compiler used to compile the Java programming instructions. Those details are supplied by the implementation of the virtual machine, and include such things as whether little Endian or big Endian format is used for storing compiled instructions, and the length of an instruction once it is compiled. Because these machine-dependent details are not reflected in the compiled code, the code can be transported to a different environment (a different hardware machine, a different operating system, etc.), and executed in that environment without requiring the code to be changed or recompiled—hence the phrase "write once, run anywhere". The compiled code, referred to as Java "bytecode", then runs on top of a JVM, where the JVM is tailored to that specific operating environment. As an example of this tailoring of the JVM, if the bytecode is created using little Endian format but is to run on a microprocessor expecting big Endian, then the JVM would be responsible for converting the instructions from the bytecode before passing them to the microprocessor.

Programs written in Java take two forms: applications and applets. Java applets are applications that are intended to be downloaded to a user's machine with a Web page, and run within the Web browser that displays the Web page. Since Java was introduced in 1995, it has gone through a number of dramatic changes in a very short period of time. During this evolution, number of advantages and disadvantages of using applications versus applets have come to light.

One of the areas of difference between applications and applets is in the Java runtime environment, as well as the affect of changes thereto. (The runtime environment includes the JVM, as well as a number of files and classes that are required to run Java application or applets. Hereinafter, the terms "JVM" and "runtime environment" will be used interchangeably unless otherwise noted.) For applets, only a single level of the JVM exists in a given version of a browser. In order to upgrade the JVM level to keep pace with changes to the language, a new version of the browser must be installed. And, as new levels of browsers are installed on client machines, developers must update and maintain the Java code, recompiling (and retesting) it to match the browser's JVM level. In addition, evolution of the Java language has in some cases resulted in functionality

(such as specific application programming interfaces, or "APIs") being deprecated between Java levels. This means that applets written in Java version 1.0.2, while they work in Java version 1.1, may not work when the browsers adopt the next version, Java 2. To continue using an applet written in an older Java version without changing the applet, an older JVM level (and therefore an older browser) must be used. While this approach solves the problem of running applets written in older Java versions, it typically does not enable deployment of new applets within this browser, because development tools typically cease supporting generation of code in the older levels. Furthermore, as defects in existing browser JVMs are identified, applet developers often create work-arounds while waiting for JVM developers to fix the problem. Once the fixes are applied, the work-arounds may cause defects in the applet. In addition, obtaining the latest release of a browser does not necessarily imply that it will provide the latest release of the JVM level, as the level of JVM within a browser tends to lag behind the currently released JVM level by 6 to 8 months. This may mean that applets under development, which will be created using a development toolkit, are created using a newer JVM level than is available in the new browser.

For applications, changes to the run-time environment are easier to deal with, as most Java applications ship bundled together with their own level of the Java runtime and those that don't state the required level of the Java runtime. However, shipping a runtime with the application means that multiple copies of the same JVM level may be installed on the client, leading to wasted storage space. When the application is not bundled with its runtime, on the other hand, the user is responsible for making sure that the correct JVM level is installed and the application is set up to use that level. Changing the runtime level so that a Java program can run, and making sure that all system settings are appropriate for the new level, is a difficult task for an end user to perform in today's environment. One solution to this problem is to write Java programs so that they will run correctly across multiple Java runtime levels. This, however, is a very difficult task for a developer, and is therefore not a viable solution.

A further issue in the run-time environment for applets is differences in how browsers from different vendors implement a particular JVM level. The browsers most commonly used today are Netscape Navigator and Internet Explorer. Because an applet developer typically has no way of predicting which browser (or browsers) will be used to run his application, good development practice calls for testing the applet with each potential browser. As will be readily apparent, the time spent testing an applet grows significantly when it is tested for multiple browsers, multiple JVM levels within each browser, etc. (as well as possibly testing implementations of the browsers on different operating system platforms). Sun Microsystems has attempted to address inter-browser differences (which also provides a way of making the latest run-time level available for applet execution) by providing a Java Plug-In which allows applets to be executed using a run-time environment provided by Sun, instead of the run-time provided by the browser. A JVM level can be selected from among those supported by the plug-in. However, this approach requires a user to understand which is the required JVM level and how to select it. In addition, the plug-in still provides a single level of a JVM until the user manually selects a different level, and therefore does not address the problems discussed above related to differences between JVM levels.

For applications, differences in JVM implementations manifest themselves differently. Typically, there is only one

version of each JVM level per operating system platform. It may be easier for a developer to predict which operating system his applications will run on than it is to predict which browser will be used for executing applets. Thus, the test and support requirements are significantly simpler for applications than for applets. Synchronization between the JVM level used in application development and the JVM level used for executing the resulting application, as well as the synchronization problems related to fixing errors, are less likely to present a problem, compared to the situation for applets that was discussed above. This is because both the development and runtime environment for applications are likely to be provided by the same vendor. In addition, when it is desirable to run an application on an older JVM (for example, due to changes such as function being deprecated, as previously discussed), this is less troublesome for an application than for an applet. The only requirement with the application scenario is that the older JVM is still available.

Another significant difference between applications and applets is their ease of use for end-users. Java-enabled browsers make it very easy for a user to run Java applets, where the user is required to do nothing more for execution than pointing the browser at the applet and clicking on a button. The user needs to know very little about the Java language and applets, and may not even realize that an applet is being invoked. Therefore, users do not need to be trained in how to run Java applets, saving time and money. Running a Java application (i.e. running a Java program outside a browser), on the other hand, is considerably more complicated. A Java application can be run from a development toolkit such as the JDK (Java Development Kit) product from Sun Microsystems; alternatively, the application may be run using the "JRE" (Java Runtime Environment) product (hereinafter, "JRE"), also from Sun Microsystems. The JRE is a subset of the JDK, providing the functionality which is required for application execution. Programs are executed from the command line when using the JRE. Running an application in either the JDK or JRE requires a fair amount of knowledge about the Java language and its environment. For example, the linked library paths and classpath environment variable must be properly set, and may change for each different application program. A number of dependencies may exist for running a particular application. For example, if the application makes use of a Java extension such as the Swing user interface components, the Swing libraries must be available. If the code for the extensions is not already installed on a user's machine, it may be difficult for an average user to locate the code and then perform a proper installation (i.e. including setting all the required variables to enable the class loader to find the code at run-time). In addition, a user must understand how to operate the JDK or JRE for program execution. While Java developers and system administrators may readily understand these types of information, it is not reasonable to place this burden on the average end-user who simply wants to execute a program.

Several problems related to differences between browser implementations have been discussed. Two additional problems are differences in support for security features, and differences in archive formats. Security features are used in an applet by invoking the security APIs provided by the browser. The primary browsers in use today have different security APIs. This forces an applet developer to write (and test) security code that is different between the browsers, increasing the cost of providing the applet code. While the "CAB" (for "cabinet") file format is used for distributing and archiving files for the Internet Explorer browser, "JAR"

(for "Java archive") file format is used to distribute and archive Java applet files.

Accordingly, a need exists for a technique by which these shortcomings in the current Java environment can be overcome. Ideally, the advantages of applets and the advantages of applications should be combined, providing an environment which then avoids the disadvantages of each. The present invention defines a novel approach to solving these problems, which will result in programs that are easier to use, and less costly to provide.

SUMMARY OF THE INVENTION

An object of the present invention is to provide a technique whereby shortcomings in the current Java environment can be overcome.

Another object of the present invention is to provide a technique whereby the advantages of applets and the advantages of applications are combined, providing an environment which then avoids the disadvantages of each.

It is another object of the present invention to provide a technique that enables dynamically switching among run-time environments for Java programs, on a per-program basis.

It is yet another object of the present invention to provide this technique in a manner that enables a user to easily switch between different run-time environments.

A further object of the present invention to provide a technique for specifying the dependencies of a Java application, including which run-time environment is required.

Yet another object of the present invention to provide this technique in a manner that enables the dependencies to be located automatically, and downloaded and installed, without requiring a static specification of location information.

Other objects and advantages of the present invention will be set forth in part in the description and in the drawings which follow and, in part, will be obvious from the description or may be learned by practice of the invention.

To achieve the foregoing objects, and in accordance with the purpose of the invention as broadly described herein, the present invention provides a method, system, and computer-readable code for use in a computing environment capable of having a connection to a network, for improving the manageability and usability of a Java environment. This technique comprises: defining a plurality of properties for a Java application, wherein the properties describe the application, zero or more extensions required for executing the application, and a run-time environment required for executing the application; and storing the defined properties along with an identification of the application. This technique may further comprise installing the application on a client machine using the stored properties. Preferably, installing the application further comprises: installing one or more dependencies of the application, wherein the dependencies comprise the required extensions and the required run-time environment; and installing a Java Archive file for the application on the client machine, and this installing dependencies further comprises: parsing the properties to locate the dependencies; determining whether each of the dependencies are already installed on the client machine; and retrieving and installing the located dependency when it is not already installed. Optionally, the technique may further comprise retrieving a latest version of the stored properties for the application prior to operation of installing the one or more dependencies. The installing one or more

dependencies may further comprise dynamically retrieving a location for use in said retrieving and installing. In one aspect, the technique may further comprise creating a registry file on the client machine corresponding to the properties. In this aspect, the technique preferably further comprises: receiving a request to execute a selected application on the client machine; constructing a proper run-time environment for the selected application using a corresponding registry file; and starting execution of the selected application in the constructed environment. The constructing may further comprise: reading the corresponding registry file to determine current dependencies of the application, wherein the current dependencies comprise currently-required extensions and a current run-time environment for the application; ensuring that each of the current dependencies of the selected application is installed; setting appropriate environment variables for the current run-time environment; and setting appropriate environment variables for the currently-required extensions. Optionally, the technique may further comprise: updating the current run-time environment in the registry file; and updating the currently-required extensions in the registry file. In addition, the technique may further comprise setting one or more parameters of the selected application using the corresponding registry file, and may provide for updating the parameters in the registry file.

The present invention will now be described with reference to the following drawings, in which like reference numbers denote the same element throughout.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a computer workstation environment in which the present invention may be practiced;

FIG. 2 is a diagram of a networked computing environment in which the present invention may be practiced;

FIG. 3 depicts the technique with which the preferred embodiment of the present invention associates properties with a Java application, and stores those properties for later use;

FIG. 4A defines the layout of the properties information used by the present invention, and FIG. 4B depicts an example of using this layout for a particular application program;

FIGS. 5A and 5B show the logic used in the preferred embodiment to locate and install dependencies for an application program, and the logic used in the preferred embodiment to install the Jobbi JAR file for an application program on a client's computer;

FIG. 6A defines the layout of the registry file used by the present invention, and FIG. 6B depicts an example of using this layout for a particular application program;

FIG. 7 depicts the logic invoked in the preferred embodiment when an application program is launched on a client computer; and

FIG. 8 depicts the logic with which run-time information may be updated after an application program has been installed.

DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. 1 illustrates a representative workstation hardware environment in which the present invention may be practiced. The environment of FIG. 1 comprises a representative single user computer workstation 10, such as a personal computer, including related peripheral devices. The work-

station 10 includes a microprocessor 12 and a bus 14 employed to connect and enable communication between the microprocessor 12 and the components of the workstation 10 in accordance with known techniques. The workstation 10 typically includes a user interface adapter 16, which connects the microprocessor 12 via the bus 14 to one or more interface devices, such as a keyboard 18, mouse 20, and/or other interface devices 22, which can be any user interface device, such as a touch sensitive screen, digitized entry pad, etc. The bus 14 also connects a display device 24, such as an LCD screen or monitor, to the microprocessor 12 via a display adapter 26. The bus 14 also connects the microprocessor 12 to memory 28 and long-term storage 30 which can include a hard drive, diskette drive, tape drive, etc.

The workstation 10 may communicate with other computers or networks of computers, for example via a communications channel or modem 32. Alternatively, the workstation 10 may communicate using a wireless interface at 32, such as a CDPD (cellular digital packet data) card. The workstation 10 may be associated with other computers in a local area network (LAN) or a wide area network (WAN), or the workstation 10 can be a client in a client/server arrangement with another computer, etc. All of these configurations, as well as the appropriate communications hardware and software, are known in the art.

FIG. 2 illustrates a data processing network 40 in which the present invention may be practiced. The data processing network 40 may include a plurality of individual networks, such as wireless network 42 and network 44, each of which may include a plurality of individual workstations 10. Additionally, as those skilled in the art will appreciate, one or more LANs may be included (not shown), where a LAN may comprise a plurality of intelligent workstations coupled to a host processor.

Still referring to FIG. 2, the networks 42 and 44 may also include mainframe computers or servers, such as a gateway computer 46 or application server 47 (which may access a data repository 48). A gateway computer 46 serves as a point of entry into each network 44. The gateway 46 may be preferably coupled to another network 42 by means of a communications link 50a. The gateway 46 may also be directly coupled to one or more workstations 10 using a communications link 50b, 50c. The gateway computer 46 may be implemented utilizing an Enterprise Systems Architecture/370 available from IBM, or an Enterprise Systems Architecture/390 computer, etc. Depending on the application, a midrange computer, such as an Application System/400 (also known as an AS/400) may be employed. ("Enterprise Systems Architecture/370" is a trademark of IBM; "Enterprise Systems Architecture/390", "Application System/400", and "AS/400" are registered trademarks of IBM.)

The gateway computer 46 may also be coupled 49 to a storage device (such as data repository 48). Further, the gateway 46 may be directly or indirectly coupled to one or more workstations 10.

Those skilled in the art will appreciate that the gateway computer 46 may be located a great geographic distance from the network 42, and similarly, the workstations 10 may be located a substantial distance from the networks 42 and 44. For example, the network 42 may be located in California, while the gateway 46 may be located in Texas, and one or more of the workstations 10 may be located in New York. The workstations 10 may connect to the wireless network 42 using the Transmission Control Protocol/

Internet Protocol ("TCP/IP") over a number of alternative connection media, such as cellular phone, radio frequency networks, satellite networks, etc. The wireless network 42 preferably connects to the gateway 46 using a network connection 50a such as TCP or UDP (User Datagram Protocol) over IP, X.25, Frame Relay, ISDN (Integrated Services Digital Network), PSTN (Public Switched Telephone Network), etc. The workstations 10 may alternatively connect directly to the gateway 46 using dial connections 50b or 50c. Further, the wireless network 42 and network 44 may connect to one or more other networks (not shown), in an analogous manner to that depicted in FIG. 2.

Software programming code which embodies the present invention is typically accessed by the microprocessor 12 of the workstation 10 and server 47 from long-term storage media 30 of some type, such as a CD-ROM drive or hard drive. The software programming code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users from the memory or storage of one computer system over a network of some type to other computer systems for use by users of such other systems. Alternatively, the programming code may be embodied in the memory 28, and accessed by the microprocessor 12 using the bus 14. The techniques and methods for embodying software programming code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein.

A user of the present invention may connect his computer to a server using a wireline connection, or a wireless connection. Wireline connections are those that use physical media such as cables and telephone lines, whereas wireless connections use media such as satellite links, radio frequency waves, and infrared waves. Many connection techniques can be used with these various media, such as: using the computer's modem to establish a connection over a telephone line; using a LAN card such as Token Ring or Ethernet; using a cellular modem to establish a wireless connection; etc. The user's computer may be any type of computer processor, including laptop, handheld or mobile computers; vehicle-mounted devices; desktop computers; mainframe computers; etc., having processing and communication capabilities. The remote server, similarly, can be one of any number of different types of computer which have processing and communication capabilities. These techniques are well known in the art, and the hardware devices and software which enable their use are readily available. Hereinafter, the user's computer will be referred to equivalently as a "workstation", "device", or "computer", and use of any of these terms or the term "server" refers to any of the types of computing devices described above.

In the preferred embodiment, the present invention is implemented as a computer software program. Availability of a network connection is assumed, which must be operable at the time when the dynamic loading software on a user's workstation is invoked. In the preferred embodiment, the present invention is implemented as one or more modules (also referred to as code subroutines, or "objects" in object-oriented programming). The server to which the client computer connects may be functioning as a Web server, where that Web server provides services in response to requests from a client connected through the Internet. Alternatively, the server may be in a corporate intranet or extranet of which the client's workstation is a component. The present invention operates independently of the communications protocol used to send messages or files between

the client and server, although the HTTP (Hyper Text Transfer Protocol) protocol running on TCP/IP is used herein as an example when discussing these message flows.

The present invention addresses a number of shortcomings in the Java environment, as will be discussed herein. The present invention also enables applets to be run without use of a browser, as if the applet was an application (and therefore all executable programs will be referred to hereinafter as "applications"). In this manner, the advantages of applets and the advantages of applications are combined. In particular, the disadvantages discussed earlier related to synchronizing applet code with the JVM level in a browser, different security APIs to invoke per browser, and the need to support multiple file archival formats are avoided by no longer using the browser as an execution environment. The acronym Jobbi—which stands for "Java code Outside the Browser By IBM"—is used herein to refer to the implementation of the present invention. Each application in Jobbi has a properties file associated with it. The information in the properties file is used to describe the requirements of an application, much as an applet tag would describe an applet's requirements in the current art. Using this properties information, each application program can specify its dependencies—including the particular runtime environment that the application should run on—as well as the environment settings that are required for running the application. Multiple run-time environments (i.e. multiple versions of a JVM or JRE) can exist on a client machine, where a single (shareable) copy of each run-time is accessible to those application programs which need it. A technique is defined herein for dynamically switching to the run-time environment which is required for a particular application, using information in the properties file. This is accomplished with little or no input from a human user. In this manner, older JVM levels are just as easily accessible for use at run-time as newer levels, freeing application developers from the need to update, recompile, and retest applications just to keep up with the moving target of the JVM level within the most recently released browser or operating system platform run-time environment.

The preferred embodiment of the present invention will now be discussed in more detail with reference to FIGS. 3 through 8.

FIG. 3 illustrates the technique with which the preferred embodiment of the present invention associates properties with a Java application, and stores those properties for later use. The format of the properties information defined by the present invention, and an example of using this information for a particular application, is shown in FIGS. 4A and 4B, respectively.

The properties definition process shown in FIG. 3 is a stand-alone process performed by an application developer, and is preferably integrated into the normal application build process which the developer uses during application development. Block 300 indicates that the developer performs this normal build process, which will use techniques that are known in the art. The output of this build process is a Java Archive file, also known as a "JAR" file, as indicated at Block 305. The present invention does not change the JAR file content. Block 310 shows the "Jobbi packager" of the present invention being invoked. This packaging step is illustrated in more detail at Blocks 320 and 325. At Block 320, the developer specifies values for the applicable properties of the application, using his knowledge about the application's requirements. These properties include application dependencies, run-time requirements, etc., as will be described in more detail below with reference to FIG. 4. A

new JAR file, designated a Jobbi JAR file, is created as a result at Block 325. This Jobbi JAR file is then stored in an archive for later use, as depicted at Block 315 of the mainline flow, in a manner similar to the way in which existing JAR files are stored. As shown in FIG. 3, the Jobbi JAR file that will be archived at Block 315 includes the JAR file created according to the prior art for the application (as indicated by element 330), and also the Jobbi properties information (indicated by element 335). From this archive, all information needed to operate the application in a Java environment is available. (Note that while FIG. 3 indicates storing both the existing archive data 330 and the Jobbi information 335 together, this is merely one technique that may be used. Alternatively, these two types of information can be separately stored without deviating from the present invention, provided that the Jobbi properties are associated with the archived application information. This association may be implemented, for example, by storing a pointer or other reference in the JAR file, which identifies the associated Jobbi properties file; or, such a pointer or reference may be stored in the Jobbi properties file, pointing back to the JAR file.)

In an alternative embodiment, the operations depicted in Block 310, 320, and 325 could be separated in time from the operation of the normal build process and JAR file creation, without deviating from the inventive concepts of the present invention. When this alternative approach is used, the JAR file created at Block 305 would be stored as in the current art. When the properties information is subsequently created for the application, the stored JAR file for the application would be located, and the properties information would then be associated or stored therewith, as described with reference to FIG. 3.

FIG. 4A defines the properties information and file layout 400 contemplated by the preferred embodiment of the present invention. The layout 400 will be explained with reference to the example 450 of FIG. 4B, which illustrates the properties information for a hypothetical application. The properties information defined in this layout provides a standardized means for packaging not only applications, but also Java run-times and Java extensions. Different types of information are pertinent to each of these types of packaged content. Ten different types of property values are shown in the layout, at elements 405, 408, 410, 415, 420, 425, 430, 435, 440, and 445. (While these ten types of information are used for the preferred embodiment, it will be obvious to one of ordinary skill in the art that additional or different values may be used in a proper setting, without deviating from the inventive concepts disclosed herein. In addition, other names may be used for the properties instead of those shown, and the order of entries may be changed from the order shown.)

Some of the information in the properties file describes the application, run-time, or extension, and other information describes its dependencies. Each individual entry will now be discussed. The first element 405 is the display name to be used for this application. The display name may be used, for example, in displaying the icon which the user will use to invoke the application. For a hypothetical application "X1", as indicated at element 455 of FIG. 4B, the display name entry syntax is shown at 459. The name of the property appears first, which in this case is "display name". The value for the property is shown as "x1's display name". For a subsequent entry 480 in the properties file, which pertains to a run-time named "Y1" (see comment entry 480), the display name 481 is illustrated as "y1's display name". (Note that in the example of FIG. 4B, the equal sign has been

used to separate the property names from their value. This is merely one separator that could be used.) Properties entry 408 specifies a display icon to be used for application files, an example of which is illustrated at 460. The value of this entry will be a file name where an icon is stored as an image, bitmap, etc. Within this property value, the "|" character is used in the preferred embodiment to indicate the path separator, and has been used in the path specification "images\hod.gif" of element 460. (This "|" symbol will be replaced on installation of the registry into the client machine, as further discussed below with reference to FIG. 6.)

The next property entry 410 is for the version of the packaged item. In the preferred embodiment, this information will be specified as a comma-separated list comprising the major, minor, revision, and build version numbers. This version information identifies which particular version of an application, run-time, or extension the properties information pertains to. For application X1 of FIG. 4B, the version syntax is shown at element 461. For run-time Y1, the version is shown at 482. Version syntax other than the comma-separated list of the preferred embodiment may be used when appropriate, to match the syntax used in a particular installation.

Property entry 415 identifies the type of information being described by this entry in the properties file. The type may be application, runtime, or extension. As shown at 462 and 483 of FIG. 4B, the keywords "application" and "runtime" have been used. Alternatively, other techniques may be used to convey the type, such as assigning numeric values (such as 0 through 2) to the packaged items.

The location type of the packaged item is specified using the property entry at 420. This location type 420 is used along with the location entry 425. As described in FIG. 4A, the location type may be "URL" (Uniform Resource Locator), in which case the location entry specifies a network location from which the archived package item can be retrieved. An example of using a URL is shown at elements 484 (where the type is identified using the "URL" keyword) and 485 (where an example URL is specified) of FIG. 4B. The location type 420 may alternatively be "file", in which case the location 425 specifies information for a directory structure in which the file is located. An example of using a file location is shown at elements 463 and 464 of FIG. 4B. In the example location shown at 464, the special character "." has been used to indicate that the archived information is in the current JAR file. In that case, there is no need to specify a location value 425 in the properties file. (While the preferred embodiment uses the special character "." to indicate this situation, other techniques such as a special keyword may alternatively be used.) When referring to archived information within a JAR file, a unique identifier or "UID" that uniquely identifies the information within the JAR file may be used. The location type 420 may also be specified as "jobbi-lookup-server", which means that the location will be determined dynamically at run-time (as will be discussed below with reference to FIG. 5B). In this case, the location value 425 will preferably be left empty. The location 425 may be specified using the keyword "prompt", as indicated at 428. In this case, the user will be prompted to enter the location information. (In the preferred embodiment, the keyword for the property type will appear in the property file, followed by the separator syntax, without an associated value when the value is empty.) As described above with reference to entry 415, shorter identifiers such as numeric values may be used instead of the keywords for entry 420.

Property entry 430 specifies whether the archived package item contains native code. This entry 430 is used in conjunction with entry 435, which specifies a string identifier of the native code platform when the value of entry 430 is "true". Examples of using these entries are shown at elements 465 and 466, where the native code value is false, and at elements 486 and 487, where the native code value is true.

Dependencies for this package item are specified using property entry 440. In the preferred embodiment, the dependency syntax uses a comma-separated list of UIDs for code that must be installed for this package item to run on any single platform. If there are no dependencies, then the value for this property will preferably be left empty. Element 467 shows an example of dependency information, where two identifiers "X2" and "Y1" are listed. As shown at elements 470 and 480, property information will also be provided for these dependencies. Element 488 shows that the example run-time Y1 has no dependencies. By specifying the dependencies within the archived package, all Java-specific information needed to install and use the package is available. For example, an application such as X1 (see element 455 of FIG. 4B) specifies the run-time environment "Y1" which it needs as a dependency at 467.

Property entry 445 defines the final entry of the preferred embodiment, which is the Java class name of the class containing the main() function. This entry has an empty value unless the package item is an application. The value listed for this entry will be, used at run-time to launch the application. As shown for application X1 at element 468, the main function for this application is located in the class "com.ibm.X1".

FIG. 5A shows the logic used in the preferred embodiment to locate and install dependencies for an application program, and FIG. 5B shows the logic used in the preferred embodiment to install the Jobbi JAR file for an application program on a client's computer. As depicted at 500, the logic of the dependency installation operates on the client's computer. This logic may be invoked in a stand-alone manner, to ensure that the dependencies for an application are installed. Alternatively, it may be invoked during execution of an application, as will be further discussed below with reference to Block 730 of FIG. 7. Dependency installation for a particular application is requested at Block 505. In the preferred embodiment, an HTTP request is sent 506 to a Web server 510, where that Web server 510 stores Jobbi JAR files 515, 516, 517, etc. which were created using the technique described above with reference to FIG. 3. This HTTP request will specify a unique identifier for the requested application. The associated Jobbi JAR file will be located by the Web server 510, and returned 507 to the client machine. Block 520 indicates that this file is received at the client machine, and is subsequently processed at Block 525. The processing of the Jobbi JAR file is explained in further detail in Blocks 535 and 540. The properties information (see FIG. 4) is parsed at Block 535 to locate the dependencies for the application requested at Block 505. When the dependency information is extracted, Block 540 checks to see whether the dependent item is installed. This checking process may have 3 outcomes, indicated at flows 541, 542, and 543. If the dependency (or dependencies) is/are already installed, then processing continues at Block 530 as indicated by flow 542. If there are dependencies, and the location of the dependency is known, then control returns to Block 505 as indicated by flow 541. As will be obvious to one of ordinary skill in the art, this is a recursive invocation of the dependency installation process. This recursive invocation may be used to retrieve the run-time needed for the requested

application, extensions needed for the application, etc. If a dependency is needed but its location is not known (for example, the value of property entry 420 is specified as "jobbi-lookup-server"), then the unique identifier of the dependency will be sent to a lookup server 545, as indicated at flow 543. This lookup server 545 will return the associated Jobbi JAR file location to the client machine, as indicated by flow 544. In the preferred embodiment, this returned information is a URL specifying the location of the server where the archived information is stored. Now that the location of the dependency is known, a recursive invocation of the dependency installation process is invoked using flow 541. This technique of determining the location of a dependency dynamically enables the present invention to provide packaging that is very flexible, as contrasted to the current art where such location information must be statically pre-specified. (Note that while a single Web server 510 is depicted in FIG. 5A, this is for illustrative purposes. More than one server may be used, where the HTTP request 506 will specify the appropriate server using its URL.)

The dependency checking process of Block 540 will be repeated for each specified dependency. The determination of whether a dependency is already installed uses techniques which are known in the art. Once the dependencies have been fully processed as described with reference to Blocks 535 and 540, control returns to the mainline processing in Block 530, where the Jobbi JAR file retrieved at 507 is installed into a Jobbi registry on the client machine. This process is described in more detail in FIG. 5B.

The installation process used to install the Jobbi JAR file for an application program on a client's computer begins at Block 570 of FIG. 5B, where the application's JAR file is extracted from the Jobbi JAR file (when the two have been stored together; see element 315 of FIG. 3) and copied to the UID directory of the client machine. (The UID directory is a directory created on the client's machine, having the same name as the UID of the Jobbi archive whose contents are contained therein. This technique facilitates finding the Jobbi archive information when subsequently setting up the environment for the application. Alternatively, other directory naming approaches may be used, in which case the name of the directory used to store the archived information for an application must be stored as part of the registry file.) Block 575 then uses the Jobbi properties information from the Jobbi JAR file, and creates a registry file from this properties information. The format of the registry file is depicted in FIG. 6, and is discussed in detail below. The registry file information 580 is then stored at Block 585 in the registry directory for this client machine. The process of FIG. 5B then ends.

FIG. 6A defines the layout 600 of the registry file used by the present invention, and FIG. 6B depicts an example 670 of using this layout for a particular application program. Whereas the properties file contains information needed to use an application on a number of platforms on which it may be installed, the registry provides tailored information about using the application on this particular client machine. At run-time, the registry information will be used to construct the proper environment for the application, as will be discussed below with reference to FIG. 7. A number of registry entries are extracted directly from the properties file information during the processing of Block 575 of FIG. 5B, as will be described herein. (In a similar manner as described with reference to the property file layout 400 in FIG. 4A, the registry file layout 600 depicts the preferred embodiment, and may pertain to an application, a runtime, or extensions. This layout information may be changed or extended in a

proper environment, it may be reordered, and other keywords may be used instead of those shown.)

The registry entry 605 specifies a unique identifier of this package, which can be any valid string. The identifier may be generated, e.g., by invoking the function of a random number generator, or by other techniques (including user input) which do not form part of the present invention. An example identifier string is shown at element 671 of FIG. 6B. Note that the registry entries have been prefixed with the syntax "jobbi" in this example: this is for illustrative purposes only, and clearly indicates that these are entries in the Jobbi registry.

Registry entry 610 is used to specify the Java class containing the main() function. This information is extracted from entry 445 of the properties file, during the processing of Block 575 of FIG. 5B, provided that the properties file is that of an application. An example of using this entry 610 is shown at element 672 of FIG. 6B, where a particular class name is specified. Registry entry 615 is the display name of this package, and can be any displayable string value. This value is extracted from properties entry 405, and is illustrated in the example 670 at element 673. Entry 620 specifies a display icon to be used for application files, an example of which is illustrated at 674. The value of this entry will be a file name where an icon is stored as an image, bitmap, etc. As indicated in the note at the bottom of FIG. 6A, the "|" character is used in the preferred embodiment to indicate the path separator, and has been used in the path specification "images\hod.gif" of element 674. This "|" symbol will be replaced on installation of the registry into the client machine, to use the "\" or "/" character, as appropriate for the client's operating system. The value of this entry 620 is extracted from element 408 of the properties file.

The relative directory or archive name which needs to be included in the classpath environment variable when this package is used is specified using registry entry 625. The value used for this entry is created automatically, and is the location in the file system of the client machine where the archived package is stored. An example is shown at 675. Note that the symbol "&" has been used, indicating that the platform-specific symbol for concatenation is to be used to replace this symbol on installation of the registry on the client's machine (as explained in FIG. 6A).

Registry entry 630 specifies a list of unique identifiers of run-time environments in which this package can be executed. This list may be initially constructed using information from the properties file, where the dependencies for the package are inspected to locate each runtime for the package. In addition, user or developer input may be used subsequently, to extend the values in this list. Element 676 shows that in this example application, any of three different run-time environments (identified in the example as "10000001", "10000002", and "10000004") may be used to execute the application.

The relative working directory which needs to be set as an environment variable is specified as the value of registry entry 635. The special syntax "." is used in the example at 677, indicating the current directory is to be used. This value is preferably created by initializing it to the value ".", and providing a means (such as a configuration menu) with which the value can subsequently be changed if needed.

The type of item described by the registry information is specified using entry 640, and will be either application, extension, or run-time. Element 678 indicates that the example pertains to an application, using the application

value of 0. The value of this entry is deduced from the properties file entry 415.

Registry entry 645 specifies the currently-selected run-time environment to be used for an application. This entry is omitted from the registry file for run-times and extensions. An example run-time identifier is shown at 679, which corresponds to the final choice from the list of run-times in element 676. The value to be used for entry 645 will be chosen from the values in entry 630. Preferably, a selection policy will be used for a particular implementation, such as choosing the final element from a list of choices, or choosing the first element, etc. Alternatively, a user may be prompted to select from the list.

The package entry 650 specifies the archive name for the package, identifying where it is stored on a server in the network using a URL or where it is stored in a file system using a file path. If the archived package has been expanded and installed into the UID directory of the client's machine, then the value of this entry is left blank, as shown by element 680.

The extem entry 655 is a semi-colon separated list of the unique identifiers on which this package is dependent, and is created from entry 440 of the properties file. When there are no dependencies, then this entry may be completely omitted from the registry file, as has been done in the example 670.

The final entry in the preferred embodiment of the registry file is the parameters entry 660. This is a list of parameters, preferably separated using semi-colons, which will be passed to the main() function upon invocation of the application. Accordingly, this entry is not specified unless the registry type 640 is application. Typically, the parameter values will be entered by a user during the application launch process, and thus no parameter values will be stored persistently in the registry file. However, it may be that one or more parameters has a somewhat constant or fixed value. In that case, the value(s) may be stored in the registry, avoiding the need to prompt the user to enter the values at run-time. The example 670 omits specification of parameter values.

FIG. 7 depicts the logic invoked in the preferred embodiment when an application program is launched on a client computer. The process begins at Block 700, when a user requests to execute an application, and takes place on the client machine (as noted at element 710). The manner in which the user requests the application does not form part of the present invention. The user may click on an icon representing the application (such as the display icon 620 identified in the registry file), select an application identifier (such as the display name 615) from a pop-up or pull-down list, invoke the application using timer-driven lists, etc. At Block 715, the process of constructing the appropriate run-time environment for the application, and starting the application executing in that environment, begins. As will be illustrated, the user is required to know little or nothing about how the run-time environment operates. Block 715 opens the registry file associated with the requested application, which contains information as described with reference to FIGS. 6A and 6B. Platform-specific logic will be invoked at Blocks 720 and 725 to process the environment data from the registry file. For example, the classdir entry 625 will be appended to the classpath variable, and the working directory will be set using information from entry 635. Block 730 then checks the dependency list 655, to ensure that all dependencies are installed. If not, then the installation process of FIG. 5 will preferably be invoked. For

15

those dependencies which are installed, a recursive invocation of the logic in FIG. 7 is performed as shown at 732, setting the appropriate information for using the dependencies. When all dependency information has been processed, control transfers to Block 735. The current run-time entry 645 of the application's registry file is extracted, and the value is used to retrieve the registry file for that run-time. Block 740 then uses information from the run-time's registry, and uses the appropriate settings for environment variables (such as appending directories to the libpath and binpath, etc.).

At Block 745, a platform-dependent JNI (Java Native Interface) is invoked. As is known in the art, the JNI is a standard, virtual machine independent interface used to enable Java applications to call native libraries of code written in other languages such as C or C++. The appropriate environment variables and application parameters are passed on this invocation, enabling Block 750 to finalize the setting of environment variables and then start the system process with the application program executing within it. The process of FIG. 7 then ends, and the program executes normally. As has been demonstrated, the novel techniques of the present invention enable the proper run-time to be used for an application, which may include changing the run-time dynamically as each different application is selected for execution.

The run-time environment for an application can be easily changed using the present invention, according to the logic depicted in FIG. 8. At Block 800, user input is entered from a graphical user interface (GUI), command line, etc., requesting to change information in the registry file. (Alternatively, means may be provided with which a systems administrator can force information updates on one or more client machines. For example, if a new run-time environment is being downloaded throughout an organization, the systems administrator may update all client registry files to use this new run-time. This approach will be useful to further reduce the amount of run-time knowledge required for the end users. Means for downloading information from a network location to client machines are known in the art, and will be used to invoke the logic of FIG. 8.) If the user request is to update the current run-time entry 645 in the registry, then Block 805 will accept the new run-time identifier from the user. Optionally, verification of this identifier may be performed. If the user request is to change persistently-stored application parameters 655, Block 810 will accept the new parameter values. Optionally, the parameter values may be verified by inspecting the applicable application to ensure that the number and type of parameter values is appropriate. If the user requests to change or add dependency information 650, then Block 815 will accept the new information. Optionally, the stored list of dependency identifiers may be presented to the user, along with means for identifying additions, deletions, and changes to this list. Once the user has entered the changed registry information, and any optional verifications have been performed, Block 820 updates the stored registry information for this application. The next time this application is launched, the revised information will be used when constructing the execution environment according to FIG. 7. Thus, it can be seen that changing an application program so that it uses a different run-time environment is greatly simplified as contrasted to the current art.

While the preferred embodiment of the present invention has been described, additional variations and modifications in that embodiment may occur to those skilled in the art once they learn of the basic inventive concepts. Therefore, it is

16

intended that the appended claims shall be construed to include both the preferred embodiment and all such variations and modifications as fall within the spirit and scope of the invention.

We claim:

1. In a computing environment capable of having a connection to a network, computer readable code readable by a computer system in said environment and embodied on one or more computer-readable media, for installing a Java application on a client machine, comprising:

a subprocess for automatically retrieving, responsive to an execution request on said client machine, a properties file for a Java application to be installed, wherein said properties file (1) describes said Java application, (2) specifies zero or more executable extensions which are required for executing said Java application, and (3) specifies a run-time environment which is required for executing said Java application;

a subprocess for installing said Java application on said client machine using said properties file; and

a subprocess for automatically installing, by said client machine, one or more dependencies of said Java application, wherein said dependencies comprise said required executable extensions and said required run-time environment, further comprising:

a subprocess for parsing said properties file to locate said dependencies; and

for each of said located dependencies which is not already installed on said client machine, a subprocess for automatically recursively (1) retrieving a properties file for said located dependency, (2) installing said located dependency, and (3) installing any dependencies identified when parsing said retrieved properties file of said located dependencies, provided said identified dependency is not already installed on said client machine.

2. Computer readable code according to claim 1, further comprising:

a subprocess for revising said properties file for said Java application, wherein said subprocess for automatically retrieving and said subprocess for automatically installing then use said revised properties file.

3. Computer readable code according to claim 1, wherein said subprocess for automatically installing one or more dependencies further comprises a subprocess for dynamically determining, for at least one of said dependencies, a location from which said at least one dependency is to be installed.

4. Computer readable code according to claim 1, further comprising:

a subprocess for creating a registry file on said client machine, wherein said created registry file contains entries corresponding to said properties file, said entries being tailored to said client machine; and

a subprocess for using said created registry to construct said run-time environment for said Java application on said client machine.

5. Computer readable code according to claim 4, further comprising:

a subprocess for receiving a request to execute a selected Java application on said client machine;

a subprocess for constructing a proper run-time environment for said selected Java application using its corresponding registry file; and

a subprocess for starting execution of said selected Java application in said constructed environment.

17

6. Computer readable code according to claim 5, wherein said subprocess for constructing further comprises:

- a subprocess for reading said corresponding registry file to determine current dependencies of said Java application, wherein said current dependencies comprise currently-required extensions and a current run-time environment for said Java application;
- a subprocess for ensuring that each of said current dependencies of said selected Java application is installed;
- a subprocess for setting appropriate environment variables for said current run-time environment; and
- a subprocess for setting appropriate environment variables for said currently-required extensions.

7. Computer readable code according to claim 6, further comprising a subprocess for setting one or more parameters of said selected Java application using values specified in said corresponding registry file.

8. Computer readable code according to claim 7, further comprising a subprocess for updating said parameters in said registry file.

9. Computer readable code according to claim 4, further comprising:

- a subprocess for updating said current run-time environment in said registry file; and
- a subprocess for updating said currently-required extensions in said registry file.

10. A system for installing a Java application on a client machine in a computing environment capable of having a connection to a network, comprising:

- means for automatically retrieving, responsive to an execution request on said client machine, a properties file for a Java application to be installed, wherein said properties file (1) describes said Java application, (2) specifies zero or more executable extensions which are required for executing said Java application, and (3) specifies a run-time environment which is required for executing said Java application;

means for installing said Java application on said client machine using said properties file; and

- means for automatically installing, by said client machine, one or more dependencies of said Java application, wherein said dependencies comprise said required executable extensions and said required run-time environment, further comprising:

means for parsing said properties file to locate said dependencies; and

for each of said located dependencies which is not already installed on said client machine, means for automatically recursively (1) retrieving a properties file for said located dependency, (2) installing said located dependency, and (3) installing any dependencies identified when parsing said retrieved properties file of said located dependencies, provided said identified dependency is not already installed on said client machine.

11. The system according to claim 10, further comprising:

- means for revising said properties file for said Java application, wherein said means for automatically retrieving and said means for automatically installing then use said revised properties file.

12. The system according to claim 10, wherein said means for automatically installing one or more dependencies further comprises means for dynamically determining, for at least one of said dependencies, a location from which said at least one dependency is to be installed.

13. The system according to claim 10, further comprising:

- means for creating a registry file on said client machine, wherein said created registry file contains entries corresponding to said properties file, said entries being tailored to said client machine; and

18

means for using said created registry to construct said run-time environment for said Java application on said client machine.

14. The system according to claim 13, further comprising: means for receiving a request to execute a selected Java application on said client machine;

means for constructing a proper run-time environment for said selected Java application using its corresponding registry file; and

means for starting execution of said selected Java application in said constructed environment.

15. The system according to claim 14, wherein said means for constructing further comprises:

- means for reading said corresponding registry file to determine current dependencies of said Java application, wherein said current dependencies comprise currently-required extensions and a current run-time environment for said Java application;

means for ensuring that each of said current dependencies of said selected Java application is installed;

means for setting appropriate environment variables for said current run-time environment; and

means for setting appropriate environment variables for said currently-required extensions.

16. The system according to claim 15, further comprising means for setting one or more parameters of said selected Java application using values specified in said corresponding registry file.

17. The system according to claim 16, further comprising means for updating said parameters in said registry file.

18. The system according to claim 13, further comprising: means for updating said current run-time environment in said registry file; and

means for updating said currently-required extensions in said registry file.

19. A method for installing a Java application on a client machine in a computing environment capable of having a connection to a network, comprising steps of:

- automatically retrieving, responsive to an execution request on said client machine, a properties file for a Java application to be installed, wherein said properties file (1) describes said Java application, (2) specifies zero or more executable extensions which are required for executing said Java application, and (3) specifies a run-time environment which is required for executing said Java application;

installing said Java application on said client machine using said properties file; and

automatically installing, by said client machine, one or more dependencies of said Java application, wherein said dependencies comprise said required executable extensions and said required run-time environment, further comprising steps of:

parsing said properties file to locate said dependencies; and

for each of said located dependencies which is not already installed on said client machine, automatically recursively (1) retrieving a properties file for said located dependency, (2) installing said located dependency, and (3) installing any dependencies identified when parsing said retrieved properties file of said located dependencies, provided said identified dependency is not already installed on said client machine.

20. The method according to claim 19, further comprising the step of:

- revising said properties file for said Java application, wherein said automatically retrieving step and said automatically installing step then use said revised properties file.

19

21. The method according to claims 19, wherein said automatically installing one or more dependencies step further comprises the step of dynamically determining, for at least one of said dependencies, a location from which said at least one dependency is to be installed.

22. The method according to claim 19, further comprising the steps of:

creating a registry file on said client machine, wherein said created registry file contains entries corresponding to said properties file, said entries being tailored to said client machine; and

using said created registry to construct said run-time environment for said Java application on said client machine.

23. The method according to claim 19, comprising the steps of:

receiving a request to execute a selected Java application on said client machine;

constructing a proper run-time environment for said selected Java application using its corresponding registry file; and

starting execution of said selected Java application in said constructed environment.

24. The method according to claim 23, wherein said constructing step further comprises the steps of:

reading said corresponding registry file to determine current dependencies of said Java application, wherein said current dependencies comprise currently-required extensions and a current run-time environment for said Java application;

ensuring that each of said current dependencies of said selected Java application is installed;

setting appropriate environment variables for said current run-time environment; and

setting appropriate environment variables for said currently-required extensions.

25. The method according to claim 24, further comprising the step of setting one or more parameters of said selected Java application using values specified in said corresponding registry file.

26. The method according to claim 25, further comprising the step of updating said parameters in said registry file.

27. The method according to claim 22, further comprising the steps of:

updating said current run-time environment in said registry file; and

updating said currently-required extensions in said registry file.

28. The method according to claim 19, wherein said Java application is a Java applet.

29. A method for improving manageability and usability of a Java environment in a computing environment, comprising steps of:

storing an identification of one or more dependencies of a Java application, wherein said dependencies comprise a run-time environment, other than a browser, which is required for executing said application and zero or more extensions required for executing said application; and

installing said Java application, wherein said installing step further comprises the step of using said stored identification to automatically locate and install said dependencies of said Java application.

20

30. A method of enabling an applet to execute outside a browser, comprising steps of:

storing information pertaining to execution of said applet, wherein said information comprises: (1) an identification of one or more permissible run-time environments in which said applet may be executed, other than said browser; and (2) an identification of zero or more executable extensions on which said applet is dependent, as well as a corresponding location from which each of said identified executable extensions may be installed; and

installing said applet using said stored information.

31. The method of enabling an applet to execute outside a browser according to claim 30, wherein said installing step further comprises steps of

ensuring that a selected one of said permissible run-time environments is available; and

ensuring that each of said identified executable extensions is installed, and installing, from the corresponding location, any of said identified executable extensions that are not already installed.

32. The method of enabling an applet to execute outside a browser according to claim 31, further comprising the step of executing said installed applet using said selected one of said permissible run-time environments and said installed executable extensions.

33. A method for executing a Java application without using a browser on a client machine in a computing environment capable of having a connection to a network, comprising steps of:

requesting, by a user, execution of a selected Java application on said client machine;

constructing, by said client machine responsive to said request, a run-time environment for said selected Java application using information retrieved from a registry file, wherein said registry file contains entries specifying values for properties of said selected Java application, said values of said entries being tailored to said client machine, further comprising steps of:

setting environment data using environment data values from said registry file;

automatically installing one or more dependencies of said selected Java application using dependency data values from said registry file, further comprising steps of:

for each of said dependencies which is not already installed on said client machine, automatically recursively (1) retrieving a properties file for said dependency, (2) installing said dependency, (3) constructing a run-time environment for said dependency using information retrieved from its registry file, and (4) installing any dependencies identified in said retrieved properties file of said dependencies, provided said identified dependency is not already installed on said client machine;

automatically locating a run-time registry file for a current run-time specified in said registry file of said selected Java application;

setting environment data using environment data values from said located run-time registry file; and

invoking a virtual-machine independent interface for calling native libraries of code in said run-time environment; and

executing said selected Java application in said constructed run-time environment.

* * * * *